

Language fundamentals

1. Identifiers
2. Reserved words
3. Data types
4. Literals
5. arrays
6. Types of variables
7. Var-arg method
8. Main method
9. Command line argument
10. Java coding standards

Identifiers: A name in the java program is called an identifier which can be used for identification purpose. It can be method name or variable name or class name or label name

Example:

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

Rules for defining java identifiers:

1. The only allowed characters in java identifiers are
 - A to Z
 - a to z
 - 0 to 9
 - \$
 - _ (underscore)
2. If you are using any other character we will get compile-time error
 - Ex: total_number,
 - total#
3. identifiers cannot starts with digit
 - ex: total123
 - 123total
4. java identifiers are case sensitive off course java language itself created as case sensitive programming language

example:

```
class Test
{
    int number=10;
    int Number=10;
    int NUMBER=10;
}
```

5. there is no length limit for java identifiers but it is not recommended to take too lengthy identifiers
6. we cannot use reserved words as identifiers
ex:
`int x=10`
`int if=20;`
7. all predefined java class names and interface names we can use as identifiers
ex:
`class Test`
`{`
`public static void main(String[] args)`
`{`
`int String=10;`
`System.out.println(String)`

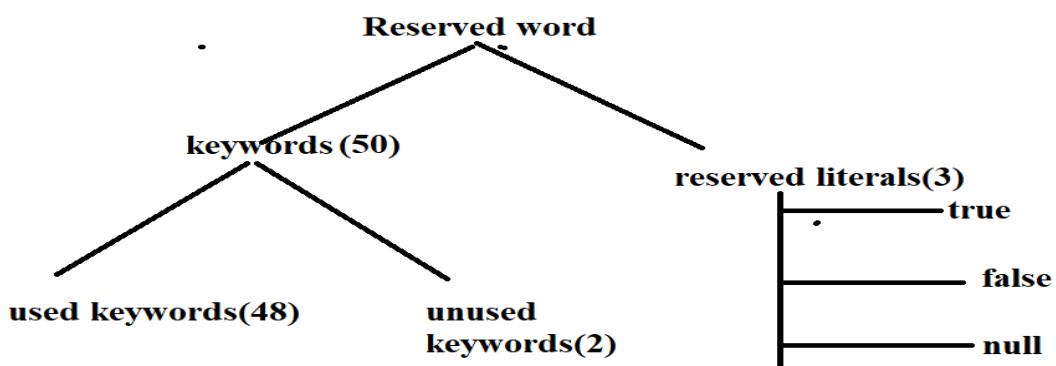
`Int Runnable=999;`
`System.out.println(Runnable);`
`}`
`}`
8. Even though it is valid but it is not a good programming practice, because it reduces readability and creates confusion

Which of the following are valid java identifiers?

total_number
total#
123total
total123
cas\$h
\$_\$\$_\$\$_
ALL@HANDS
Integer
int
Int

Reserved word:

In java, some reserved to represent some meaning or functionality such type of words are called reserved words



Keywords for data types (8):

Byte
Short
Int
Long

Float
Double
Boolean
Char

Key word for flow control (11):

If
Else
Switch
Case
Default
While

Do
For
Break
Continue
Return

Keyword for modifiers (11):

Public
Private
Protected
Static
Final
Abstract

Synchronized
Native
strictfp (1.2v)
transient
volatile

Keyword for exception handling (6):

Try
Catch
Finally

Throw
Throws
Assert (1.4v)

Class related keywords (6):

Class
Interface
Extends

Implements
Package
Import

Object related keywords(4):

New
Instanceof

Super
This

Void return type keyword:

Void

In java return type is mandatory if a method won't return anything then we have to declare that method with void return type.

But in c language return type is option and default return type is “ int ”

Unused keywords:

goto: usage of goto created several problems in old languages and hence sun people banned this keyword in java

const: use of ‘final’ instead of ‘const’

Note: goto and const are unused keywords and if we are trying to use we will get compile time error

Reserved literals (2):

“true” and “false” values for Boolean data types

“null” default value for object reference

“enum” keyword(1.5 v):

We can use enum to define a group of named constants

```
enum months
{
    Jan, feb, march, ..... dec;
}
```

Enum beer

```
{ 
    Kf, ko, rc, fo;
}
```

Conclusions:

1. all 53 reserved words in java contains only lower case alphabet symbols
2. in java we have only new keyword and there is no delete keyword because destruction of useless objects is the responsibility of garbage collector
3. the following are new keyword in java
4. strictfp - 1.2v
5. assert – 1.4v
6. enum – 1.5v
7. strictfp but not strictFp
8. instanceof but not instanceof
9. synchronized but not synchronize
10. extends but not extend
11. implements but not implement
12. const but not constant
13. import but not imports

Which of the following list contains java reserved words ?

1. new, delete
2. goto, constant
3. break, continue, return exit
4. final, finally, finalize,
5. throw, throws, thrown
6. notify, notifyAll
7. implements, extends, imports
8. sizeof, instanceof
9. instanceOf, strictFp
10. byte, short, Int
- 11. none of the above**

Which of the following are java reserved words?

public
static
void
main
String
Args

Data types

In java every variable and every expression has some type. Each and every data type is clearly defined and every assignment should be checked by compiler for type compatibility. Because of above reasons we can conclude java language is strongly typed programming language.

Java is not considered as pure objected oriented programming language because several oops features satisfied by java (like operator overloading, and multiple inheritance et..) More over we are depending on primitive data types which are non objects.

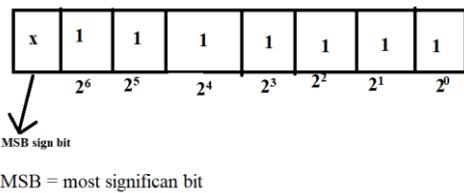
Primitive data types (8)

1. Numeric data types
 - a. Integral data types
 - i. Byte
 - ii. Short
 - iii. Int
 - iv. Long
 - b. Floating point data types
 - i. Float
 - ii. Double
2. Non- numeric data types
 - i. Char
 - ii. Boolean

Except Boolean and char remaining consider as signed data types because we can represent both positive and negative numbers

Byte:

Size: 1 byte (8 bits)
Max_value: +127
Min_value: -128
Range: -128 to +127



The most significant bit act as sign bit

0 means positive number
1 means negative number

Positive numbers will be represented in directly in the memory whereas negative numbers will be represented in two's complimented form.

byte b=10

byte b=127

byte b=128 **compatable error: possible loss of precision, found: int, required:byte**

byte b=10.5 **compatable error: possible loss of precision, found: double, required:byte**

byte b=true **compatable error: possible loss of precision, found: boolean, required:byte**

byte b= "durga" **compatable error: possible loss of precision, found: java.lang.String , required:byte**

Byte is the best choice if we want to handle data in terms of streams either from the file or from the network(file supported form or network supported form byte)

Short data type:

This is the most rarely used data type in java

Size: 2 bytes (16 bits)

Range: -2^{15} to 2^{15-1} (-32768 to 32767)

Short s=32767

shorts s=32768 **compatible error: possible loss of precision, found: int, required: short**

short s=10.5 compatible error: possible loss of precision, found: double, required: short

short s=true; **compatible error: possible loss of precision, found: Boolean, required: short**

Short data type is best suitable for 16 bit processor like 8085 but these processors are completely outdated and hence corresponding short data type is also outdated data type

Int data type:

The most commonly used data type in java is int

Size : 4 bytes(32 bits)

Range: -2^{31} to 2^{31-1} (-2147483648 to 2147483647)

int x = 2147483647

int x=2147483648 **comiletime error: integer number too large**

int x=2147483648L **comiletime error: possible loss of precision, found: long, required:int**

int x = true; **comiletime error: possible loss of precision, found: boolean, required:int**

Long data type:

Some times may not enough to hold big values then we should go for long type

Example 1: the amount of distance travelled by light in 1000 days, to hold this value int may enough we should go for long data type

Long l= 126000*60*60*24*1000 miles;

Example 2: the number characters present in big file may exceed int range hence the return type of length method Is long but not int

Example: Long l= f.length()

Size = 8bytes (64 bits)

Range: -2⁶³ to 2⁶³-1

Note: all the above data types (byte, short, int, long) ment for representing integral values if we want to represent floating point values then we should go for floating point datatypes

Floating data types:

Float:

If we want 5 to 6 decimal places of accuracy then we should go for float

Float fallows single precision

Size: 4 bytes

Range: -3.4e38 to 3.4e38

Double:

If we want 14 to 15 decimal places of accuracy then we should go for double

Double fallows double precision

Size 8 bytes

Range: -1.7e308 to 1.7e308

Boolean data type:

Size: NA (vm dependent)

Range: NA(but allowed values are: true/ false)

Example:

boolean b= true

boolean b = 0 ; **ce: incompatible types found: int, required: Boolean**

boolean b = True; **ce: cannot find symbol, symbol:variable True, location: class test**

boolean b = "True"; **ce: incompataible types, found: java.lang.String, required: boolean**

Example:

Int x=0;

If(x) → **ce: incompatible types, found: int, required: boolean**

{

System.out.println("hello");

}

Else{

System.out.println("hi")

}

Example:

```
while(1) → ce: incompatible types found: int, required: boolean
{
System.out.println("hello")
}
```

Char data type:

Old languages (like c or c++) are ASCII code based and the number of allowed different ASCII are <=256 To represent this 256 characters 8 bits are enough hence the size of char in old languages is 1 byte

But java is Unicode based and the number of different characters are >256 and <=65536 to represent these many characters 8 bits may not enough compulsory we should go for 16bits hence the size of char in java is 2 bytes

Size: 2 byte

Range: 0 to 65535

Summary of java primitive data types:

Data type	Size	range	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	-3.4e38 to 3.4e38	Float	0.0
Double	8 bytes	-1.7e308 to +1.7e308	Double	0.0
Boolean	NA	NA(but allowed values are true/false)	Boolean	False
char	2 bytes	0 to 65535	Character	0(represents space character)

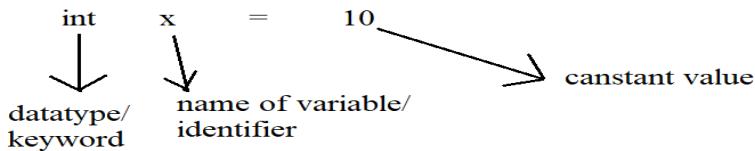
Note: null is a default value for object reference and can not apply for primitives, if we are trying to use for primitive then we get compile time error

Example:

```
Char ch=null ce: incompatible types, found: null type, required: char
```

Literals

A constant value which can be assigned to the variable is called the literal



Integral literals:

For integral data types (byte, short, int long) we can specify literal value in the following ways

Decimal literals (base -10):

allowed digits are 0-9

Example: int x=10;

Octal form(base – 8):

allowed digits are 0 – 7 literal value should be prefixed with zero (0)

Example: int x=010

Hexa decimal form(base – 16):

allowed digits are 0 - 9, a – f

For extra digit (a – f) we can use both lower case and upper case characters, these is one of very few areas where java is not case sensitive, the literal value should be prefixed with 0x or 0X

Example: int x= 0x10

Int x= 0X10

These are only possible to specify literal value for integral data types

Which of the following declarations are valid?

Example:

Int x=10;
Int x==0786; ce: integer number too large
Int x=0777;
Int x=0XFace;
Int x=0XBeef;
Int x= 0XBeer; invalid

Class Test

```
{  
    Public static void main(String[] args)  
    {  
        Int x=10;  
        Int y =010;  
        Int z= 0x10;  
        System.out.println(x+ “..”+y+ “....”+z);  
    }  
}
```

$(10)_8=(?)_{10}$ $0*8^0+1*8^1=8$ $(10)_{16}=(?)$ $0*16^0+1*16^1=16$ Out put : 10.8....16

By default every integral literal is of int type but we can explicitly as long type by suffixed with small l or capital L

```
int x =10;  
int l=10L  
int x =10L; ce: possible loss of precision, found: long, required: int  
long l=10;
```

There is no direct way to specify byte and short literals explicitly but indirectly we can specify whenever we are assigning integral literal to the byte variable and if the value within the range of byte then compiler treats it automatically as byte literal similarly short literal also

```
Byte b=10  
Byte b=127;  
Byte b=128; ce: possible loss of precision , found: int, required:byte  
  
Short s= 32767;  
Short s=32768; ce: plp found: int; required: short
```

Floating point literals:

By default every floating point literal is of double type and hence we can not assign directly to the float variable

But we can specify floating point literal as float type by suffixed with 'f' or 'F'

Example:

```
Float f=123.456; CE: possible loss of precision, found: double, required: float;  
Float f=123.456F;  
Double d= 123.456;
```

We can specify explicitly floating point literal as double type suffixed with d or D off course this convention is not required

```
Double d=123.456D;  
Float f= 123.456d; CE: possible loss of precision, found=double, required= float;
```

We can specify floating point literals only in decimal form and can not specify in octal and hexa decimal forms.

Example:

```
Double d= 123.456  
Double d=0123.456 → it is a decimal literal but not octal literal  
Double d=0X123.456 → CE: malformed floating point literal
```

We can assign integral literal directly to floating point variables and that integral literal can be specified either in decimal, octal and hexa decimal forms

Example:

```
Double d=0786; CE: integer number too large
```

```
Double d=0XFace;  
Double d=0786.0;  
Double d=0XFace.0  
Double d=10;  
Double d=0777;
```

We can't assign floating point literals to integral types

Example:

```
double d=10;
```

```
int x=10.0;CE: plp, found: double, required: int;  
We can specify floating point literal even in exponential form (scientific notation)  
double d=1.263;  
Sop(d); 1200.0
```

```
float f=12e3; CE: possible loss precision, found: double; required: float  
Float f=1.2e3F
```

Boolean literals:

The only allowed values for Boolean data type are “true” or “false”.

Boolean b= true;

Boolean b= 0; CE: incompatible types; found: int, required: boolean

Boolean b= True; CE: cannot find symbol; symbol: variable true; location: class test;

Boolean b=”true” CE: incompatible types; found: java.lang.String. Required: Boolean;

Example 1:

```
int x=0;  
If(x) CE: incompatible types; found : int; required: Boolean  
{  
    System.out.println("hello");  
}  
Else  
{  
    System.out.println("hi");  
}
```

Example 2:

```
While(1) CE: incompatible types; found : int; required: Boolean  
{  
System.out.println("hello");  
}
```

Char literals:

We can specify char literal as single character with in single quotes.

Char ch= ‘a’;

Char ch=a; CE: cannot find symbol; symbol : variable a; location: class test;

Char ch= “a” CE: incompatible types ; found: java.lang.String , required: char

Char ch= ‘ab’;

CE1: unclosed char literal

CE2: unclosed char literal

CE3: not a statement

We can specify char literal as integral literal which represents Unicode value of the character and that integral specified either in decimal, octal and hexa decimal forms but allowed range is 0 to 65535

Example:

Char ch=97;
Sopln(ch); → a

Charr ch= 0XFace;

www.unicode.org

Char ch=0777;

Char ch=65535;

Char ch =65536; **CE: possible loss of precision, found: int, required: char;**

We can represent char literal in Unicode representation which is nothing but '\uxxx' (4 digit hexadecimal number)

Example:

Char ch='\u0061'
Sopln(ch); → a;

Every escape character is valid char literal

Example:

Char ch='\n';
Char ch='\t';
Char ch='\m'; **CE: illegal escape character**

Escape character	Description
\n	New line
\t	Horizontal tab
\r	Carriage return
\b	Back space
\f	Form feed
\'	Sigle quote
\\"	Double quote
\\\	Back slash

Which of the following are valid?

Char ch= 65536;
Char ch= 0XBeer;
Char ch=\uface;
Char ch='ubeef';
Char ch='\m';
Char ch ='iface';

String literal:

Any sequence of characters with in double quotes is treated as string literal

Example:

String s= "durga";

1.7 version enhancement with respect to literals:

binary literals: for integral data types until 1.6 version we can specify literal value in the following ways

Decimal form

Octal form

Hexa decimal form

But 1.7 version onwards we can specify literal value even in binary form also allowed digits are 0 and 1. Literal value should be prefixed with 0b or 0B

Example:

Int x= 0B1111;

Sopln(x); →15

Usage of underscore(_) symbol in numeric literals:

from 1.7 version onwards we can use underscore symbol between digits of numeric literal.

Example:

Double d=123456.789 → double d=1_23_456.7_8_9
→ double d= 123_456.7_8_9

The main advantage this approach is readability of the code will be improved

At the time of compilation these underscore symbols will be removed automatically hence after compilation the above lines will become

Double d=123456.789

We can use more than one underscore symbol also between the digits

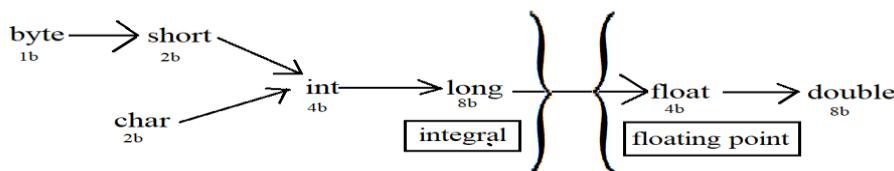
Example:

Double d=1_23_4_5_6.7_8_9
Double d=1_2_3_4_5_6.7_8_9

We can use underscore symbol only between the digits if we are using anywhere else we can get compile time error

Example:

Double d=_1_23_456.7_8_9;
Double d=1_2_3_456_.7_8_9;
Double d=1_23_456.7_8_9_;



Note: 8-byte long value we can assign to 4-byte float variable because both are following different memory representation internally

Float f= 10L;
Sopln(f); → 10.0

Arrays

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

Introduction:

An array is an index collection of fixed number of homogeneous data elements.

The main advantage of arrays is we can represent huge number of values by using single variable so that readability of the code will be improved

But the main disadvantage of array is fixed in size that is once we create an array there is no chance of increasing or decreasing the size based on our requirement hence to use arrays concept compulsory we should know the size in advance, which may not possible always

Array declaration:

One dimensional array declaration:

int[]	x;	→ recommended because name is clearly separated from type
int	[]x;	
int	x[];	

At the time of declaration we can not specify the size otherwise we will get compile time error

int [6]	x; invalid
int[]	x; valid

Two dimensional array declaration:

Int[][]	x; → valid
Int	[][]x; → valid
Int	x[][];--. valid
Int []	[]x; → invalid
Int []	x [];→ invalid
Int	[]x[]; → in valid

Which of the following are valid?

Int []	a,b; →a→1 →b→1
Int []	a[],b; →a→2 →b→1
Int []	a[], b []; →a→2 →b→2
Int []	[]a,b; →a→2 →b→2
Int []	[]a, b[]; →a→2 →b→3
Int[]	[]a,[]b; CE

If we want to specify dimension before the variable that facility is applicable only for first variable in a declaration

If we are trying to apply for remaining variable we will get compile time error

Int [] []a,[]b, []c;

Three dimensional array declaration:

Int[] [] []	a;
Int	[] [] []a;
Int	a[] [] [];
int[]	[] []a;
int[]	a[] [];
int[]	[]a[];
int[] []	[]a;
int[] []	a[];
int	[] []a[];
int	[]a[] [];

Array creation:

One dimensional array:

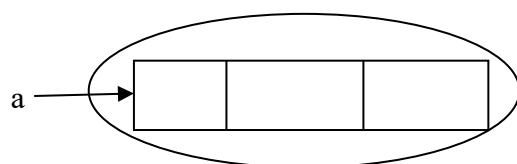
Every array in java is an object only hence we can create arrays by using **new** operator

int[] a= new int[3];

for every array type corresponding class are available and these classes are part of java language and not available to the programmer level

int[] a= new int[3];

System.out.println(a.getclass().getname()); →[I



Array Type	Corresponding class name
int[]	[I
int[][]	[I
double[]	[D
short[]	[S
byte[]	[B
boolean[]	[Z

At the time array creation compulsory, we should specify the size otherwise we will get compile time error

Example:

Int[] x= new int[]; invalid

Int[] x= new int [6]; valid

It is legal to have an array with size zero in java

Example:

Int[] x= new int[0];

If we are trying to specify array size with some negative int value then we will get run time exception saying neagative array size exception

Example:

Int[] x = new int[-3]; RE: negative array size exception

To specify array size allowed data types are byte, short, char and int.if we are trying to specify any other type then we will get compile time error

Example:

Int[] x= new int[10];

Int[] x= new int['a'];

Byte b=20;

Int[] x= new int[b];

Short s=30;

Int[] x= new int[s];

Int[] x= new int[10L]; CE: plp, found: long, required: int

The maximum allowed array size in java Is 2147483647 which is the maximum value of int data type

Example:

Int[] x= new int[2147483647];

Int[] x= new int[2147483648]; CE: integer number too large

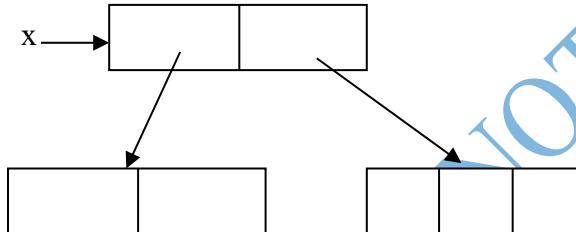
Even in the first case we may get runtime exception if sufficient heap memory not available

Two-dimensional array creation:

In java two dimensional array not implemented by using matrix style, sun people fallowed array of arrays approach for multi dimensional array creation
The main advantage of this approach is memory utilization will be improved

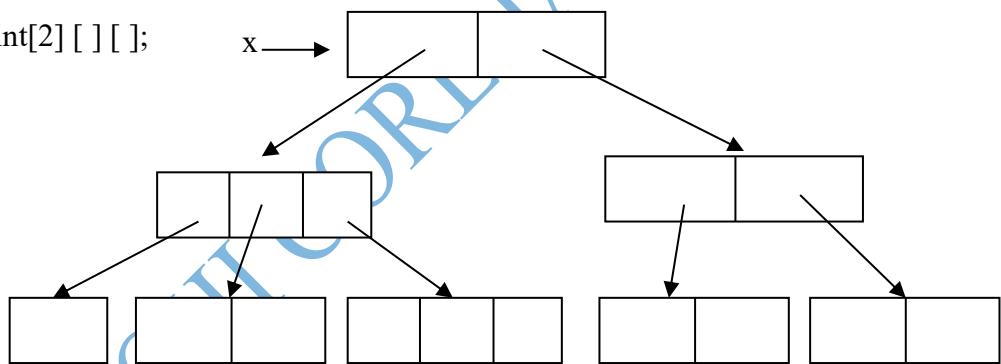
Example 1:

```
Int [ ] [ ] x= new int[2][ ];  
X[0] = new int[2];  
X[1]= new int[3];
```



Example 2:

```
Int [ ] [ ] [ ] x = new int[2] [ ] [ ];  
X[0]=new int[3][ ];  
X[0][0]=new int[1];  
X[0][1]=new int[2];  
X[0][2]=new int[3];  
X[1]= new int[2][2];
```



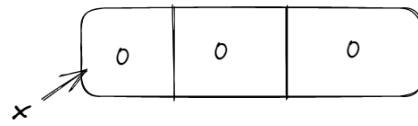
Which of the following array declaration are valid ?

- Int [] a= new int[];
- Int [] a= new int[3];
- Int [] [] a= new int [] [];
- Int [] []a= new int[3] [];
- Int [] []a= new int [] [4];
- Int [] []a= new int[3] [4];
- Int [] []a= new int [3] [4] [5];
- Int [] [] []a= new int[3] [4] [];
- Int [] [] []a= new int [3] [] [5];
- Int [] [] [] a= new int[] [4] [5];

Array initialization:

Once we create array every element by default initialized with default value.

```
Int[ ] x= new int[3];  
System.out.println(x); [I@3e256  
System.out.println(x[0]); 0
```

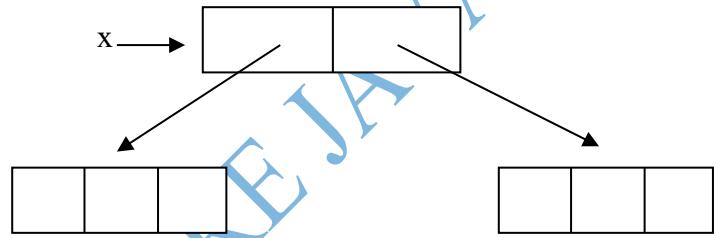


Note: Whenever we are trying to print any reference variable internally to string method will be called which is implemented by default to return the string in the following form

```
class name@hashcode _ in _ hexadecimalform
```

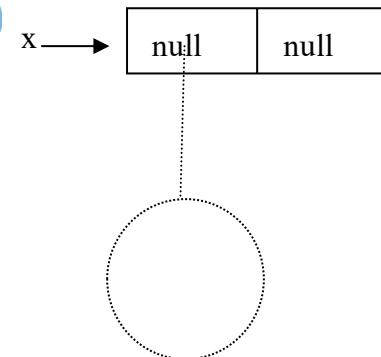
Example2:

```
int [ ] [ ]x= new int[2][3];  
sopln(x); [[I@3e25a5  
sopln(x[0]);[I@19821f  
sopln(x[0][0]); 0
```



Example3:

```
int [ ] [ ]x= new int[2][];  
sopln(x); [[I@3e25a5  
sopln(x[0]);null  
sopln(x[0][0]); RE: null pointer exception
```



note: If we are trying to perform any operation on null then we will get runtime exception saying null pointer exception.

Once we create an array every array element by default initialized with default values, if we are not satisfied with default values then we can overwrite these values with our customized values

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

```
X[0]=10;  
X[1]=20;  
X[2]=30;  
X[3]=40;  
X[4]=50;  
X[5]=60;
```

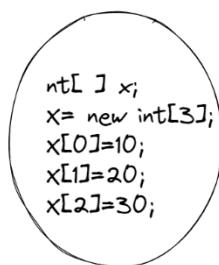
```
X[6]=70; RE: array index out of boundsexception  
X[-6]=80; RE: array index out of boundsexception  
X[2.5]=90; CE: PLP fount: double, required: int;
```

0	0	0	0	0	0	0	0	0
10	20	30	40	50	60	70	80	90

Note: if we are trying to access array element without of range index(either positive or negative int value) then we will get runtime exception saying array index out of bounds exception

Array declaration, creation and initialization in a single line:

We can declare create and initialization an array in a single line (shortcut representation)



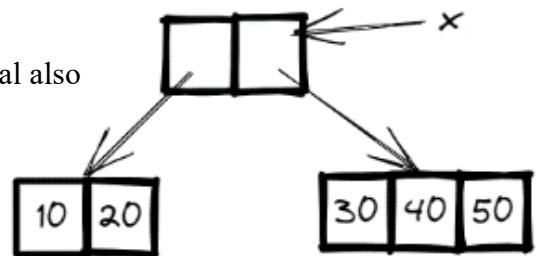
```

int [ ] x={10,20,30};
char[ ] ch= {'a', 'e', 'I', 'o', 'u'};
string s= {"A","AA","AAA"};

```

We can extend this shortcut for multi-dimensional also

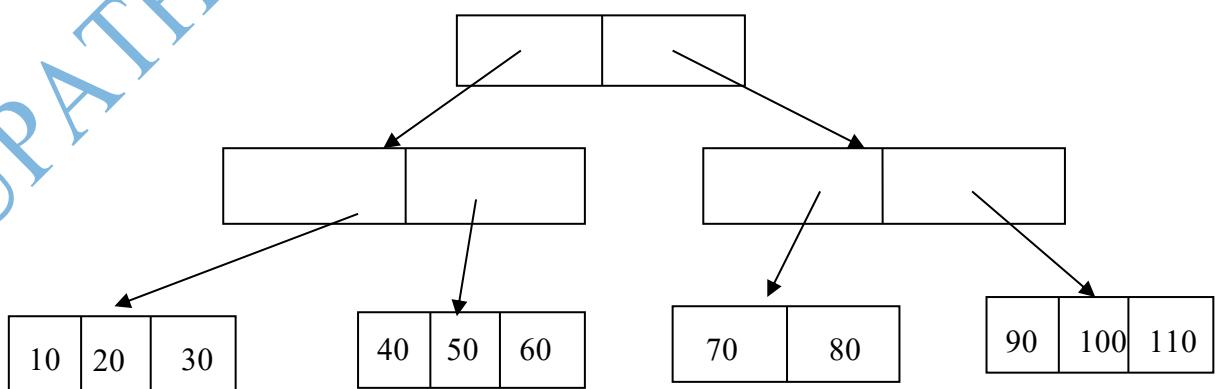
```
Int[ ][ ] x={{10,20},{30,40,50}};
```



```

Int [ ][ ][ ] x ={{{{10,20,30},{40,50,60}},{ {70,80},{90,100,110}}}
Sopln(x[0][1][2]); →60
Sopln(x[1][0][1]); → 80
Sopln(x[2][0][0]); → RE: array integer out of boundary exception
Sopln(x[1][2][0]); → RE: array integer out of bounds exception
Sopln(x[1][1][1]); → 100
Sopln(x[2][1][0]); → RE: array integer out of bounds exception

```



If we want to use this shortcut compulsory we should perform all activities in a single line, if we are trying into multiple lines then we will get compile time error.

```
Int[ ] x= {10,20,30};  
Int [ ] x;  
X={10,20,30} CE: illegal start of expression;
```

Length vs length():

Length:

Length is a final variable applicable for arrays.

Length variable represents the size of the array.

Example:

```
Int[ ] x = new int[6];  
Sopln(x.length()); → CE: cannot find symbol; symbol: method length(); location:class in[ ]  
Sopln(x.length()); → 6
```

Length():

Length method is a final method is applicable for string objects

Length method returns number of characters present in the string.

Example:

```
String s= "malachi";  
Sopln(s.length());  
→ CE: cannot find symbol; symbol: variable length; location: class java.lang.String;  
Sopln(s.length()); → 5
```

Note: length variable applicable for arrays but not for string objects whereas length method applicable for string objects but not for arrays.

```
String [ ] s= {"a","aa","aaa"};  
Sopln(s.length()); → 3  
Sopln(s.length()); → CE: cannot find symbol; symbol: method length();location: class String  
Sopln(s[0].length()); → CE: cannot find symbol, symbol: variable length; location: class  
j.l.String  
Sopln(s[0].length()); → 1
```

In multi dimensional array length variables represents only base size but not total size.

```
Int[ ] [ ] x= new int[6][3];  
Sopln (x.length()); → 6  
Sopln(x [0].length()); → 3
```

There is no direct way to find total length of a dimensional array, but indirectly we can find as follows

```
X[0].length+x[1].length+x[2].length+.....
```

Anonymous arrays:

Sometimes we can declare arrays without name such type of names less arrays are called anonymous arrays.

The main purpose of anonymous arrays is just for instant use (one time usage)

We can create anonymous array as follows

New int[] {10,20,30,40}

While creating anonymous arrays we cannot specify the size other wise we will get compile time error.

New int[3]{10,20,30} → invalid

New int[]{10,20,30} → valid

We can create multi dimensional anonymous arrays also

New int [] []{{10,20},{30,40,50}}

Based on over requirement we can give the name for anonymous array then it sis no longer anonymous

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

Example:

Class Test

```
{  
    Public static void main(String[ ] args)  
    {  
        Sum(new int[ ]{10,20,30,40}) →anonymous array  
    }  
    Public static void Sum(int[ ] x)  
    {  
        Int total =0;  
        For(int x1: x)  
        {  
            Total = total+x1;  
        }  
        System.out.println("the sum:"+ total);  
    }  
}
```

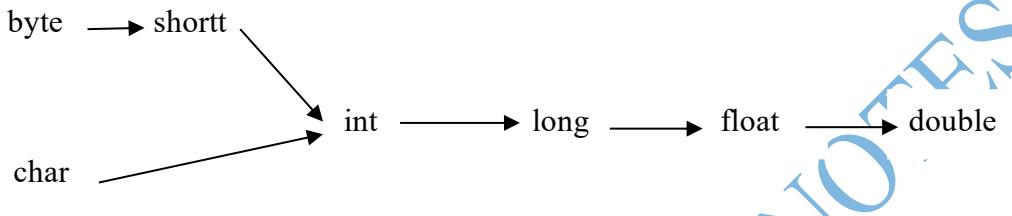
In the above example just to call sum method we required an array but after completing sum method call we are not using that array any more, hence for this one time requirement anonymous array is the best choice.

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

```
Int[ ] x= new int [5];
X[0]=10;           byte → shortt
X[1]='a';
Byte b=20;
X[2] =b;
Short s=30;
X[3]=s;
X[4]=10L; → CE: PLP found: long; required:int
```



Example 2:

In the case float type arrays the allowed data types are byte, short, char, int, long and float

Case- 2:

In the case of object type arrays as array elements we can provide either declared type objects or its child class objects.

Example 1:

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

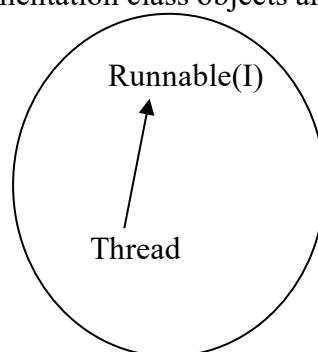
Example 2:

```
Number[ ] n= new Number[10];
N[0]= new Integer(10);
N[1]= new Double(10.5);
N[2]= new String("malachi");
→ CE: incompatible types; found: java.lang.String; required: J.S.Number
```

Case- 3:

For interface type arrays as array elements its implementation class objects are allowed

Example:
Runnable[] r= new Runnable[10];
R[0] = new Thread();
R[1]= new String("durga");
CE: incompatible types
Found: java.lang.String;
Required : java.lang.Runnable



Array type	Allowed element types
Primitive arrays	Any type which can be implicitly promoted to declared type
Object type arrays	Either declared type or its child class objects
Abstract class type arrays	Its child class objects
Interface type array	Its implementation class objects are allowed

Array variable assignments:

Case -1:

Element level promotions are not applicable at array level. For example char element can be promoted to int type whereas char array cannot be promoted to int array.

Example:

```
Int[ ] x= {10,20,30,40}
Char[ ] ch= {'a','b','c','d'};
Int[ ] b=x;
Int[ ] c= ch; → CE: incompatible types; found: char[ ]; required: int[ ];
```

Which of the following promotions will be performed automatically?

Char	→	int	valid
Char[]	→	int[]	invalid
Int	→	double	valid
Int[]	→	double[]	invalid
Float	→	int	invalid
Float[]	→	int[]	invalid
String	→	Object	valid
String	→	Object[]	valid

But in the case object type arrays child class type array can be promoted to parent class type array

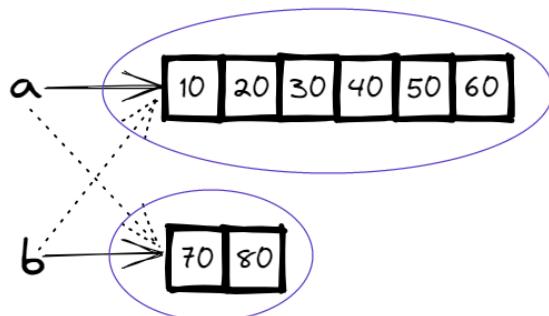
Example:

```
String[ ] s={"a","b","c"};
Object[ ] a=s;
```

Case -2:

When ever we are assigning one array to another array internal elements won't be copied just reference variable be re-assigned

```
Int[ ] a= {10,20,30,40,50,60};  
Int [ ] b= {70,80}  
A=b→ valid  
B=a→ valid
```



Case -3:

Whenever we are assigning one array to another array the dimension must be matched for example in the place one dimensional int array we should provide one dimensional only if we are trying to provide any other dimension then we get compile time error

```
Int[ ] [ ] a= new int[3][ ];  
A[0] = new int[4][3]; → CE: incompatible types; found: int[ ] [ ]; required: int[ ];  
A[0] =10; →CE: incompatible types; found:int; required int[ ]  
A[0] = new int[2]→ valid
```

Note: whenever we are assigning one array to another array both dimension and types must be matched but size are not required to match

Example 1:

```
Class Test{  
    Public static void main(String[ ] args){  
        For (int i=0;i<args.length;i++){  
            System.out.println(args[i]);  
        }  
    }  
}
```

Out put: Java test a,b,c A B C RE: array index out of bound exception Java test a b A B RE: array index out of bound exception Java test RE: array index out of bound exception
--

Example 2:

```
Class Test{  
    Public static void main(String[ ] args){  
        String[ ] argh={"x", "y", "z"};  
        Args=argh;  
        For (String s:args){  
            System.out.println(s);  
        }  
    }  
}
```

Out put: Java test a,b,c X Y z Java test a b X Y z Java test X Y z
--

Example 3:

Int[][] a = new int [4][3]; → 5

A[0]= new int [4]; → 1

A[1] = new int[2];→1

A= new int[3][2];→4

Total how many objects created?

Ans: 11

Total how many objects eligible for garbage collection?

Ans: 7

Types of variables

Division- 1:

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

Primitive variables: can be used to represent primitive values

Example:

int x= 10;

Reference variables: can be used to refer objects

Example:

Student s= new Student();

Division- 2:

Based on position of declaration and behaviour all variables are divided into three types

Instance variables.

Static variables.

Local variables.

Instance variables:

1. If the value of a variable is varied from object to object such type of variable are called instance variables.
2. For every object a separate copy of instance variable will be created.
3. Instance variable also known as **object level variables or attributes**
4. Instance variable should be declared within the class directly but outside of any method, block and constructor.
5. Instance variable will be created at the time of object creation and destroyed at the time of object destruction, hence the scope of instance variable is exactly same as the scope of object
6. Instance variable will be stored in the heap memory as the part of object.
7. We cannot access instance variable directly from static area but we can access by using object reference.
8. But we can access instance variable directly from instance area

Example:

```
Class Test{  
    Int x=10;  
    Public static void main(String[] args)  
    {  
        System.out.println(x);  
    }  
    Public void m1()  
    {  
        System.out.println(x);  
    }  
}
```

→ CE: non- static variable x cannot be reference from a static context

```
Test t= new Test();  
System.out.println(t.x); → 10
```

9. For instance variables JVM always provide default values and we are not required to perform initialization explicitly

Example:

```
Class Test  
{  
    Int x;  
    Double d;  
    Boolean b;  
    String s;  
    Public static void main(String [ ] args)  
    {  
        Test t1= new Test();  
        System.out.println(t1.x); 0  
        System.out.println(t1.d); 0.0  
        System.out.println(t1.b); false  
        System.out.println(t1.s); null  
    }  
}
```

Static variables:

1. If the value of a variable is not varied from object to object then it is not recommended to declare variable as instance variable we have to declare such type of variable at class level by using static modifier
2. In the case of instance variable 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 the class.
3. Static variables should be declared with in the class directly but outside of any method, block, constructor.
4. Static variables will be created at the time class loading and destroyed at the time class unloading hence scope of static variable is exactly same as scope of dot(.) class file

Java test
Start JVM
Create and start main thread
Locate Test.class file
Load Test.class → static variables creation
Execute main() method
Unload Test.class → static variables destruction
Terminate main thread
Shut down JVM

Static variable will be stored in **method area**

we can access static variables either by object reference or by class name but recommended to use class name within the same class it is not required to use class name and we can access directly

example:

```
class Test
{
    Static int x=10;
    Public static void main(String[ ] args)
    {
        Test t= new Test();
        System.out.println(t.x);
        System.out.println(Test.x);
        System.out.println(x);
    }
}
```

we can access static variable directly from both instance and static areas

example:

```
class Test
{
    Static int x=10;
    public static void main(String[ ] args)
    {
        System.out.println(x);      valid
    }
    public void m1()
    {
        System.out.println(x);      valid
    }
}
```

for static variables JVM will provide default values and we are not required to perform initialization explicitly

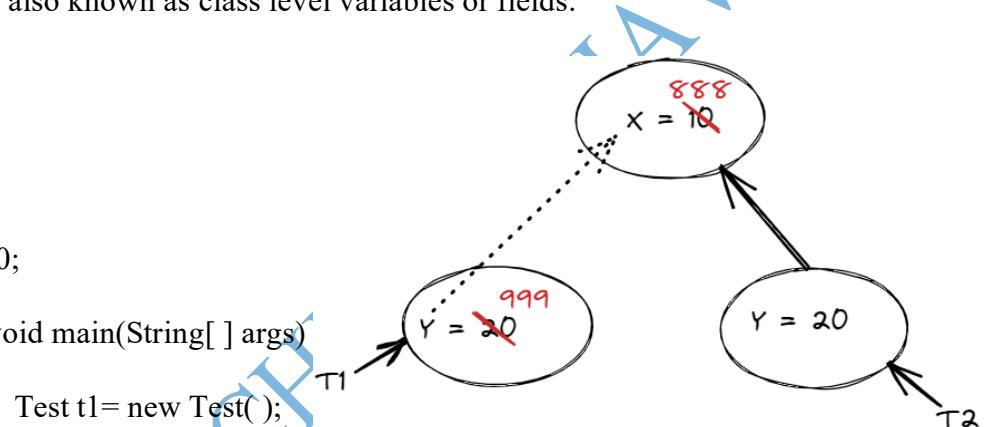
Example 2:

```
Class Test
{
    Static int x;
    Static double d;
    Static String s;
    Public static void main(string[ ] args)
    {
        System.out.println(x); 0
        System.out.println(d); 0.0
        System.out.println(s); null
    }
}
```

static variable also known as class level variables or fields.

Example:

```
Class Test
{
    Static int x=10;
    Int y=20;
    Public static void main(String[ ] args)
    {
        Test t1= new Test();
        T1.x=888;
        T2.y=999;
        Text t2= new Test();
        System.out.println(t2.x+ "----"+ t2.y); → 888 ---- 20
    }
}
```



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 temporary variables or stack variables or automatic variables.

Local variables stored inside a **stack memory**.

Local will be created while executing the block in which we declared it, once block execution is complete automatically local variable will be destroyed hence the scope of local variables is the block in which we declared it

Example:

```
Class Test
{
    Public static void main(String[] arg)
    {
        Int i=0;
        For(int j=0;j<3;j++)
        {
            I=i+j;
        }
        System.out.println(i+ "-----" +j);
        → CE: cannot find symbol;
          symbol: variable j; location class test
    }
}
```

Example 2:

```
Class Test
{
    Public static void main(String[] args)
    {
        Try
        {
            Int j= Integer.parseInt("ten");
        }

        Catch(NumberFormatException e)
        {
            J=10;
        }
        System.out.println(j);
    }
}
```

CE: cannot find symbol
 Symbol: variable j
 Location: class test

Note: for local variables JVM won't provide default values compulsory we should perform initialization explicitly before using that variable that is we are not using then it is not required to perform initialization

Example:

```
Class Test
{
    Public static void main(String[ ] args)
    {
        Int x;
        System.out.println("hello");
    }
}
```

o/p: hello

Example 2:

```
Class Test
{
    Public static void main(String[] args)
    {
        Int x;
        System.out.println(x); →CE: variable x might not have been initialization
    }
}
```

Example 3:

```
Class Test
{
    Public static void main(String [ ] arg)
    {
        Int x;
        If(args.length>0)
        {
            X=10;
        }
        System.out.println(x); →CE: variable x might not have been initialization
    }
}
```

Example 4:

```
Class Test
{
    Public static void main(String [ ] arg)
    {
        Int x;
        If(args.length>0)
        {
            X=10;
        }
        Else
        {
            X=20;
        }
        System.out.println(x);
    }
}
```

o/p:
java test A B
o/p: 10
java test
o/p 20

Note:

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

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

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

Example:

Class Test

```
{  
    Public static void main(String[ ] args)  
    {  
        Public int x=10;  
        Private in t x=10;  
        Protected int x=10;  
        Static int x=10;  
        Transient int x=10;  
        Volatile int x=10  
    }  
    Final int x=10;  
}  
}
```

CE: illegal start of expression

Note: if we are not declaring with any modifier then by default it is a default modifier but this rules is applicable only for instance and static variables but not for local variables.

Conclusions:

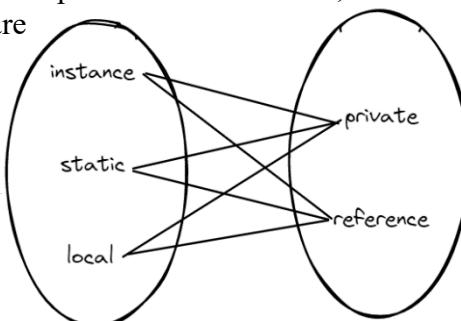
For instance and static variables JVM will provide default values and we are not required to perform initialization explicitly but for local variables JVM won't provide default values compulsory we should perform initialization explicitly before using that variable.

Instance and static variables can be accessed by multiple thread simultaneously and hence these are not thread safe but in the case of local variables for every thread separate copy will be created and hence local variables are thread safe.

Types of variable	Is thread safe or not?
Instance variables	No
Static variables	No
Local variables	yes

Every variable in java should be either instance or static or local.

Every variable in java should be either primitive or reference, hence various possible combinations of variables in java are



Example:

```

class Test
{
    Int x = 10;           → instance -- primitive
    Static String s= " malachi"; → static -- reference
    Public static void main(Striing[ ] args)
    {
        Int[ ] y = new int[j]; → local -- reference
    }
}

```

Uninitialized arrays:

Example -1:

```

Class Test
{
    Int[ ] x;
    Public static void main(String[ ] args)
    {
        Test t= new Test();
        System.out.println(t.x); null
        System.out.println(t.x[0]); RE: null pointer exception
    }
}

```

Instance level

Example 2:

```

Int[ ] x;
System.out.println(obj.x); → null
System.out.println(obj.x[0]); → RE: null pointer exception

```

Example 3:

```

Int[ ] x= new int[3];
System.out.println(obj.x); → [I@e332sas
System.out.println(obj.x[0]); → 0
Static level

```

Example 4:

```

Static Int[ ] x;
System.out.println(x); → null
System.out.println(x[0]); → RE: null pointer exception

```

Example 5:

```

Static Int[ ] x= new int[3];
System.out.println(x); → [I@e332sas
System.out.println(x[0]); → 0

```

Local level

Example 4:

```
Int[ ] x;  
System.out.println(x); → CE: varaiable x might not have been initialized  
System.out.println(x[0]); → CE: varaiable x might not have been initialized
```

Example 5:

```
Int[ ] x= new int[3];  
System.out.println(x); → [Ie@332sas  
System.out.println(x[0]); → 0
```

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

Var – arg methods(varaibale number of arugments methods)

Until 1.4 version we cannot declare a method with variable number of arguments if there is a change in number of arguments compulsory we should go for new method it increases length of the code and reduced the readability to overcome this 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 fallows.

```
m1( int... x)
```

we can call this method by passing any number int values including '0' number

example:

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

Example:

```
class Test  
{  
    Public static void m1(int... x)  
    {  
        System.out.println("var-arg method");  
    }  
    Public static void main(String[ ] args)  
    {  
        m1();  
        m1(10);  
        m1(10,20);  
        m1(10,20,30,40);  
    }  
}
```

o/p :
var-arg method
var-arg method
var-arg method
var-arg method

Internally var-arg parameter will converted into one dimensional array hence within the var-arg method we can differentiate values by using index
Example:

```
class Test{  
{  
    Public static void main(String[ ] args)  
    {  
        sum( );  
        sum(10,20);  
        sum(10,20,30);  
        sum(10,20,30,40);  
    }  
    Public static void sum(int... x)  
    {  
        int total = 0;  
        for(int x1:x)  
        {  
            total= total+x1;  
        }  
        System.out.println("the sum:="+total);  
    }  
}
```

O/P:
The sum= 0
The sum= 30
The sum= 60
The sum= 100

Case- 1:

Which of the following are valid var- arg method declarations?

- | | |
|--------------|-----------|
| m1(int... x) | → valid |
| m1(int ...x) | → valid |
| m1(int...x) | → valid |
| m1(int x...) | → invalid |
| m1(int ..x) | → Invalid |
| m1(int .x..) | → invalid |

Case- 2:

We can mix var-arg parameter with normal parameter

Example:

```
M1(int x,int... y);  
M1(String s, double... y);
```

Case- 3:

If we mix normal parameter with var-arg parameter then var-arg parameter should be last parameter

Example:

```
M1(double... d, String s); → invalid  
M1(char ch, String... s); → valid
```

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

M1(int... x, double... d); invalid

Case- 4:

Inside a class we can't declare var-arg method and corresponding one dimensional array methods simultaneously other wise we will get compile time error

Example:

Calss Test

```
{  
    Public static void m1(int... x)  
    {  
        System.out.println("int...");  
    }  
    Public static void m1(int[ ] x)  
    {  
        System.out.println("int[ ]");  
    }  
}
```

→ CE: cannot declare bot m1(int[]) and m1(int...) int test

Case- 5:

Example:

Class Test

```
{  
    Public static void m1(int... x)  
    {  
        System.out.println("var-arg method");  
    }  
    Public static void m1(int x)  
    {  
        System.out.println("general method");  
    }  
    Public static void main(String[ ] arg)  
    {  
        M1();      var-arg method  
        M1(10,20); var- arg method  
        M1(10);    general method  
    }  
}
```

In general var-arg method will get least priority that is if no other method matched then only var-arg method will get chance it is exactly same as default case in side switch.

Equivalence between var-arg parameter and one-dimensional array

Case -1:

Where ever one dimensional present we can replace with var-arg parameter

M1(int[] x) → m1(int... x)

Example:

Main(String[] args) → main(String... args)

Case -2:

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

M1(int... x) → m1(int[] x) → invalid

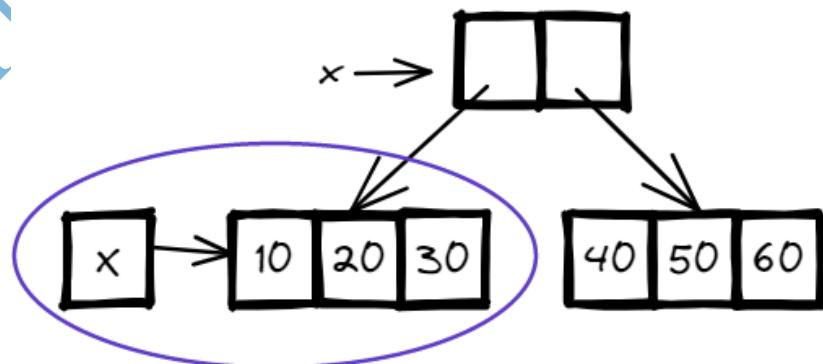
Note:

m1(int... x) we can call this method by passing a group of int values and x will become one dimensional array.

m1(int[]... x) we can call this method by passing a group of one dimensional int arrays and x will become two dimensional int array.

Class Test

```
{  
    Public static void main(String[ ] arg)  
    {  
        Int[ ] a={10,20,30};  
        Int[ ] b={40,50,60};  
        M1(a,b);  
    }  
    Public static void m1(int[ ]... x)  
    {  
        For(int[ ] x1:x)  
        {  
            System.out.println(x1[0]);  
        }  
    }  
}
```



Main() method

whether class contains main method are not and whether main method is declared according to requirement or not these things won'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 run time exception saying “ **no such method error: main** ”.

example:

javac Test.java → valid its compile

java Test

RE: no such method error:main

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

public static void main(String[] args)

public: to call by JVM from anywhere

static: without existing object also JVM has to call this method.

void: main() method won't return anything to JVM

main(): this is the name which is configured inside JVM

String[] arg: command line arguments

The above syntax is very strict and if we perform any change then we will get run time exception saying “ **No Such method error: main** ”.

Even though above syntax is very strict the following changes are acceptable

Instead of **public static** we can take **static public** that is the order of modifier is not important

We can declare String[] in any acceptable form.

Example:

main(String[] args)

main(String []args)

main(String args[])

Instead of **args** we can take any valid java identifiers.

Example:

main(String[] malachi)

We can replace String[] with var-arg parameter

Example:

main(String... args)

We can declare main method with the following modifiers

Final

Synchronized

Stricfp

Example:

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

Which of the following main method declaration are valid?

- | | |
|--|-----------|
| public static void main(String args) | → invalid |
| public static void Main(String[] args) | → invalid |
| public void main(String[] args) | → invalid |
| public static int main(String[] args) | → invalid |
| final synchronized strictfp public void main(String[] args) | → invalid |
| final synchronized strictfp public static void main(string[] args) | → valid |
| public static void main(String... args) | → valid |

in which of the above cases we will get compile time error?

We won't get compile time error anywhere but except last two cases in remaining we get runtime exception saying "**no such method error: main**".

Case-1:

Over loading of the main method is possible but JVM will always call String array argument main method only the other overloaded method we have to call explicitly like normal method call

Example:

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

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

Overloaded
methods

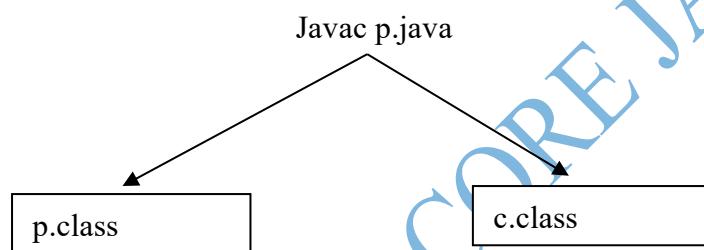
Output:
String[]

Case-2:

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

Example:

```
Save  
with  
p.java {  
  
class P  
{  
    public static void main(String[ ] args)  
    {  
        System.out.println("parent main");  
    }  
}  
class C extends P  
{  
}
```



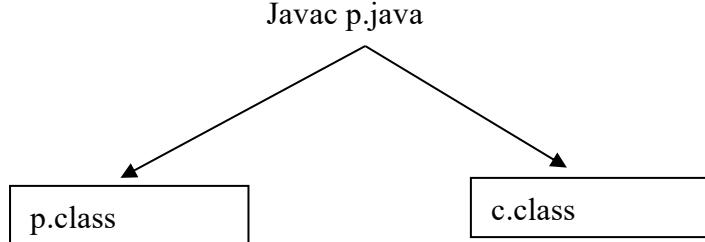
Output:
Java p
Parent main
Java c
Parent main

Case-3 :

Example:

It is method
hiding but not
overriding

```
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");  
    }  
}
```



Output:
Java p
Parent main
Java c
child main

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

Note: For main method inheritance and overloading concepts are applicable but overriding concept is not applicable instead of overriding method hiding is applicable.

1.7 version enhancements with respect to main method:

Until 1.6 version if the class doesn't contain main method then we will get runtime exception saying "no such method error: main".

But 1.7 version onwards instead of "no such method error" we will get more elaborated error information

Example:

```
class Test  
{  
}
```

version:

```
javac Test.java  
java Test  
Runtime error: NoSuchMethodError:  
main
```

version:
java c Test.java
java Test
Error: main method not found in class
Test, please define the main method as:
public static void main(String[] args)

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

Example:

```
class Test  
{  
    static  
    {  
        System.out.println("static block");  
    }  
}
```

1.6 version:
javac Test.java
java Test
o/p: static block
Runtime error: NoSuchMethodError:
main

1.7 version:
java c Test.java
java Test
Error: main method not found in class Test,
please define the main method as:
public static void main(String[] args)

Example- 2:

```
class Test
{
    static
    {
        System.out.println("static block");
        System.exit(0);
    }
}
```

1.6 version:

```
javac Test.java
java Test
o/p : static block
```

1.7 version:

```
java c Test.java
java Test
Error: main method not found in class
Test, please define the main method as:
public static void main(String[ ] args)
```

Example- 3:

```
class Test
{
    static
    {
        System.out.println("static block");
        System.exit(0);
    }
}
```

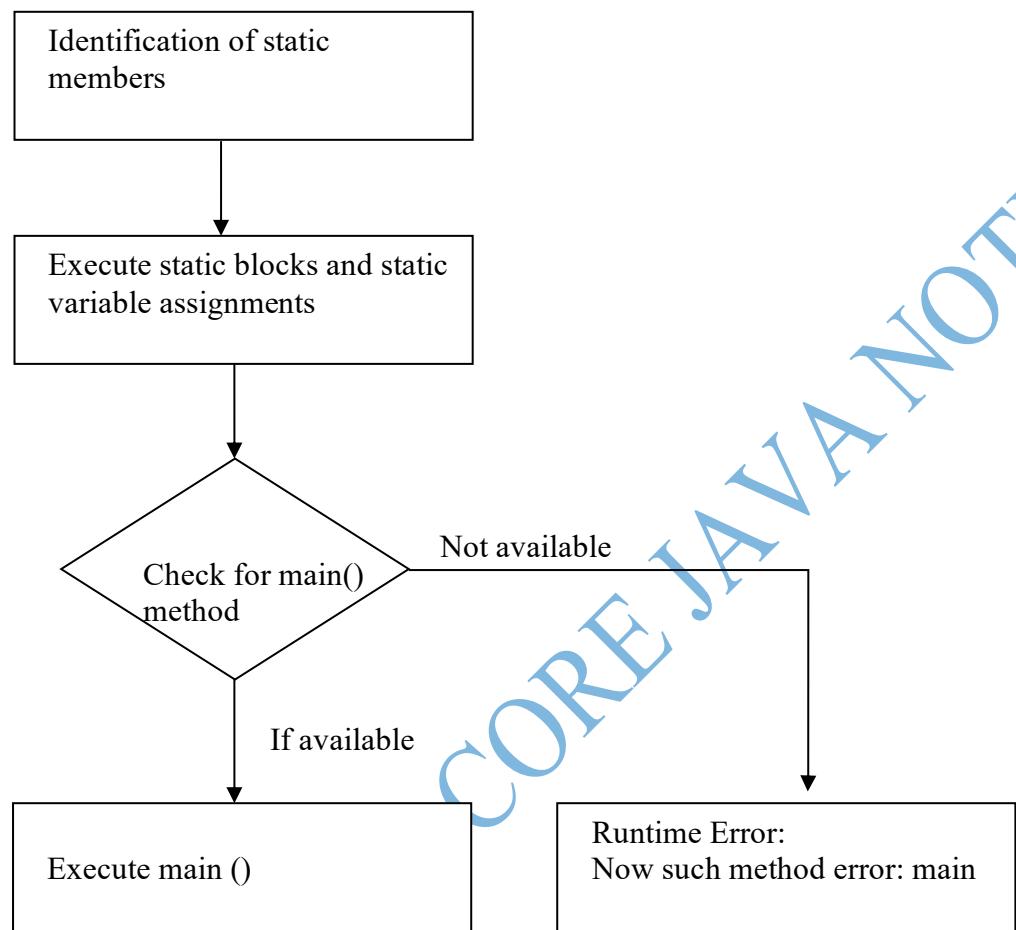
1.6 version:

```
javac Test.java
java Test
o/p : static block
```

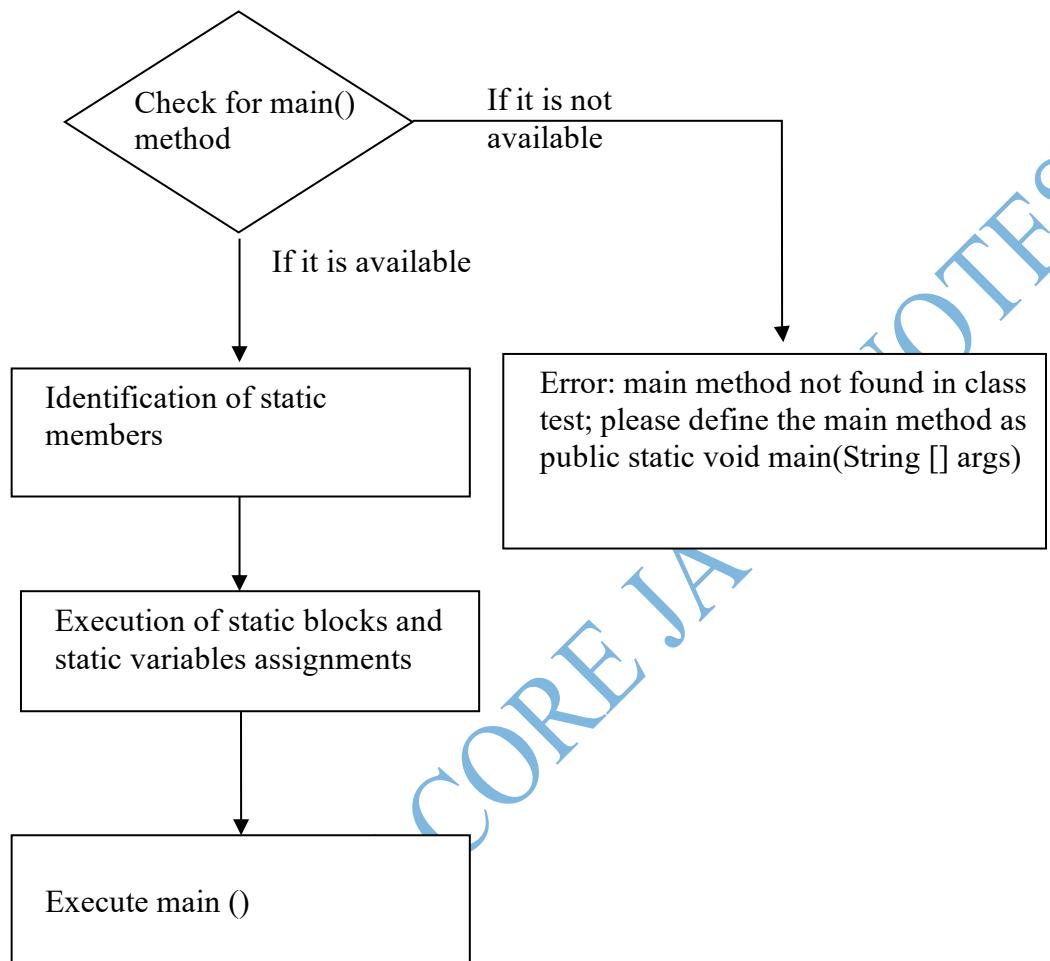
1.7 version:

```
java c Test.java
java Test
output:
static block
main method
```

1.6 version flow:



1.7 version flow:



Without writing main method is it possible to print some statement to the console?

Ans: Yes, by using static block but this rule is applicable until 1.6 version but 1.7 version onwards it is impossible to print some statements to the console without writing main method

Command Line arguments

The arguments which are passing from command prompt are called command line arguments, with these command line arguments JVM will create an array and by passing that array as argument JVM will call main method.

Example:

```
Java Test A B C;  
Args[0]  
Args[1]  
Args[2]  
Args.length→ 3
```

The main objective of command line arguments we can customize behaviour of the main method.

Example-1:

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

If we replace '<=' with '≤' then we won't get any runtime exception

Output:

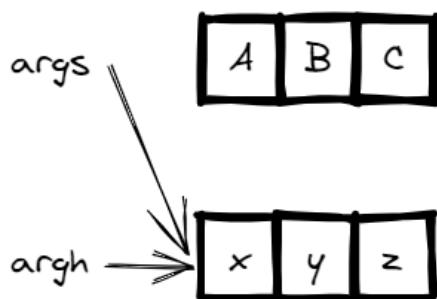
```
Java Test A B C;  
A  
B  
C  
RE: array index is out of bound  
exception
```

```
Java Test A B ;  
A  
B  
RE: array index is out of bound  
exception
```

```
Java Test ;
```

Example-2:

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



Output:

Java Test A B C;

X

Y

Z

Java Test A B ;

X

Y

Z

Java Test ;

X

Y

Z

Example – 3:

Within main method command line arguments are available in string form

```
class Test
{
    Public static void main(String[ ] args)
    {
        System.out.println(arg[0]+args[1]); →1020
    }
}
```

Output:

Java Test 10 20;

1020

Example – 4:

Usually space itself is the separator between command line arguments if our command line argument itself contain a space we have to enclose that command line argument with double quotes

```
class Test
{
    public static void main(String[ ] args)
    {
        System.out.println(arg[0]); → note book
    }
}
```

Output:

Java Test “note book”;

Java coding standards

Whenever we are writing 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 main advantage of this approach is readability and maintainability may be improved.

Example:

Class A

```
{  
    Public int m1(int x,int y)  
    {  
        Return x+y;  
    }  
}
```

Ameerpeta standard style

Package com.dugasoft.scjp;

Public class calculator

```
{  
    Public static int add(int number1, int number2)  
    {  
        Return number1+number2;  
    }  
}
```

Hi-Tech standard style

Coding standard for classes:

Usually class names are nouns should starts with upper case character and if it contains multiple words every inner word should start with uppercase character

Example:

String, StringBuffer, Account, Dog

Coding standards for interfaces:

Usually interface names are adjectives should starts with uppercase character and if it contains multiple words every inner word should starts with uppercase character

Example: Runnable, Serializable, Comparable

Coding standards for methods:

Usually method name are either verbs or verb- noun combination should starts with lower case alphabet symbol and if it contains multiple words then every inner word should start with upper case character(camel case convention)

Example: print(), sleep(), run(), eat(), stat()
getName(), setName()

Coding standards for variable:

Usually variables names are nouns should with lower case alphabet symbol and if it contains multiple words then every inner word should start with upper case character(camel case convention)

Example: name, age, salary, mobileNumber;

Coding standards for constants:

Usually constant name are nouns should contain only upper case characters and if it contains multiple words then these are words are separated with underscore symbol.

Example: MAX_VALUE, MAX_PRIORITY,NORM_PRIORITY, MIN_PRIORITY, PI

Note: usually we can declare constants with public static and final modifiers

Java bean coding standards:

A java bean is a simple java class with private properties and public getter and setter methods

Example:

```
Public class StudenBean
{
    Private String name;
    Public void setName(String name)
    {
        This.name=name;
    }
    Public String getName()
    {
        Return name;
    }
}
```

Note :Class name ends with bean is not official convention from sun

Syntax for setter method:

- It should be a public method
- The return type should be void
- Method name should be prefixed with set
- It should take some argument that is it should not be no argument method

```
Public void setName(String name)
{
    This.name=name;
}
```

Syntax for getter method:

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

```
Public String getName()
{
    Return name;
}
```

Note: ** for Boolean properties getter method name can be prefixed with either get or is but recommended to use is

```
Private Boolean empty;
Public Boolean getEmpty(){
    Return empty
}
Public Boolean isEmpty(){
    Return empty;
}
```

Coding standards listeners:

Case 1: To register listeners

Method name should be prefixed with add

Example:

```
public void addMyActionLlisterner(MyActionListener l)
public void registerMyActionListerner(MyActiionListener l)
public void addMyActionListener(Actionlistener l)
```

Case 2: To un-register listeners

Method name should be prefixed with remove

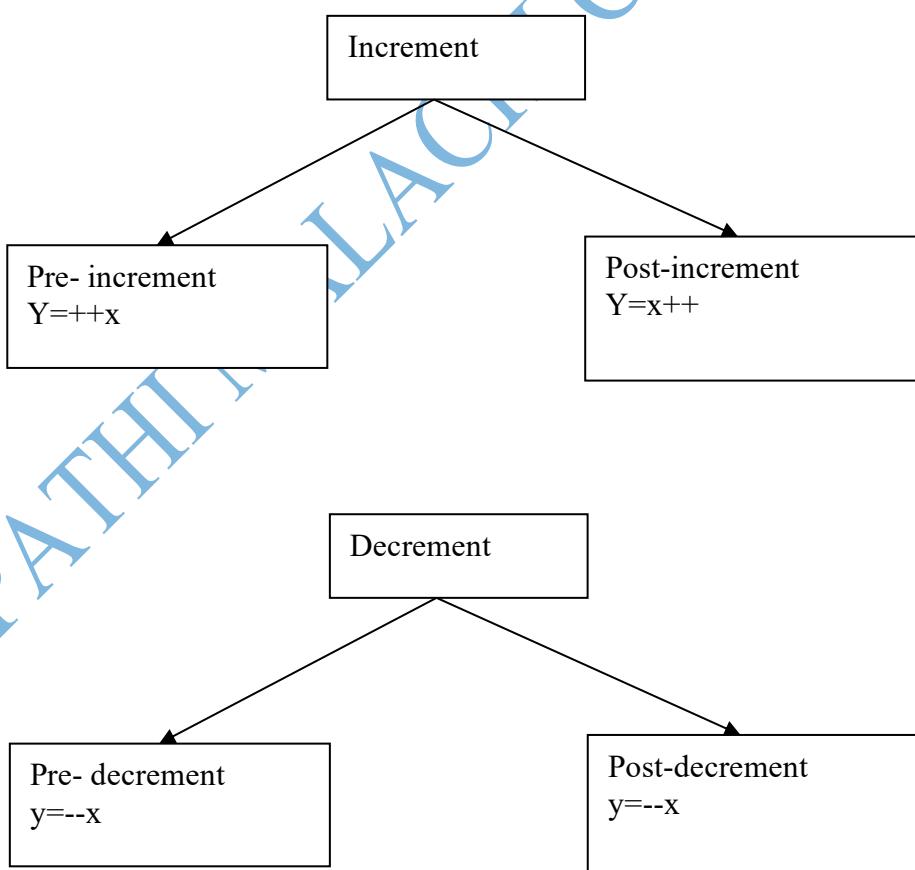
Example:

```
public void removeMyActionLlisterner(MyActionListener l)
public void unregisterMyActionListerner(MyActiionListener l)
public void removeMyActionListener(Actionlistener l)
public void delete MyActionListener(myActionListener l)
```

Operators and assignments

1. increment and decrement operators
2. arithmetic operators
3. String concatenation operator
4. relational operators
5. equality operators
6. instanceof operator
7. bitwise operators
8. short circuit operators
9. type cast operator
10. assignment operators
11. conditional operator
12. new operator
13. [] operator
14. operator precedence
15. evaluation order of operands
16. new vs newInstance()
17. instanceof vs isInstance()
18. ClassNotFoundException vs NoClassDefFoundError

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

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 compile time error

Example:

```
int x=10;
int y=++x;
System.out.println(y); → 11
```

Example :

```
int x=10;
int y=++10; → CE: unexpected :
value; required: variable
System.out.println(y);
```

Listing of increment and decrement operators not allowed

Example:

```
int x = 10
int y = ++(++x); → CE: unexpected type; required: variable, found: value
System.out.println(y);
```

For final variables we can't apply increment and decrement operators.

Example:

```
final int x=10;
X++; → CE: can not assign a value to final variables x
System.out.println(x);
```

We can apply increment and decrement operators for every primitive type except Boolean.

Example:

```
int x=10;
x++;
System.out.println(x); → 11
```

Example:

```
double d=10.5;
d++;
System.out.println(d); → 11.5
```

Example:

```
char ch='a';
ch++;
System.out.println(ch); → b
```

Example

```
boolean b = true;
b++;
System.out.println(b); → CE:
operator ++ cannot applied to
boolean
```

Difference between “b++” and b= “b+1”:

If we apply any arithmetic operator between two variables A and B the result type is always

max(int, type of A , type of B)

Example:1

```
byte a=10;  
byte b=20;  
byte c=a+b; → CE: possible loss of precision; found: int; required : byte  
System.out.println(c);
```

```
byte c= (byte)(a+b)  
System.out.println(c); → 30
```

Example: 2

```
byte b=10;  
b=b+1; → CE: possible loss of precision, found: int, required: byte  
b=(byte)(b+1); → valid  
System.out.println(b);
```

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

b++ means b = (type of b)(b+1);

Example:

```
byte b=10;  
b++;  
System.out.println(b); → 11
```

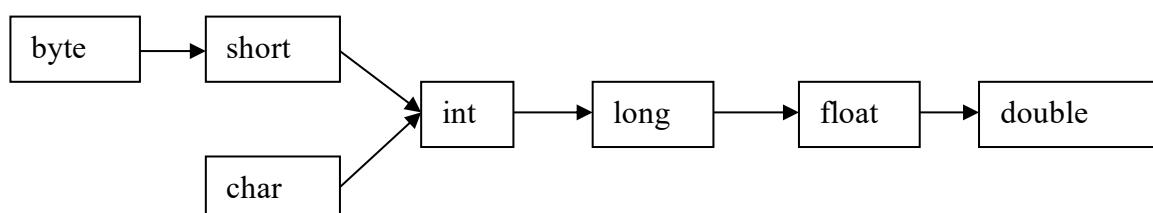
Arithmetic operators(+, -, *, /, %) :

If we apply any arithmetic operator between two variables A and B the result type always max(int, type of a, type of b)

byte + byte = int
byte + short = int
short + short = int
byte + long = long
long + double = double
float + long = float
char + char = int
char + double = double

Example:

```
System.out.println('a'+'b'); → 195 (a=97 b=98)  
System.out.println('a'+0.89 ); → 97.89(a=97)
```



Infinity:

in integral arithmetic (byte, short, int, long) there is no way to represent infinity hence if infinity is the result we will get arithmetic exception in integral arithmetic

System.out.println(10/0); → RE: arithmetic exception / by zero

But in floating point arithmetic (float and double) there is a way to represent infinity for this Float and Double classes the following two constants

POSITIVE_INFINITY;

NEGATIVE_INFINITY;

Hence even though the result is infinity we won't get any arithmetic exception in floating point arithmetic

System.out.println(10/0.0); → infinity

System.out.println(-10.0/0); → -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 run time exception saying arithmetic exception

System.out.println(0/0); RE: arithmetic exception / by zero

But in floating point arithmetic(float and double) there is a way to represent undefined the results for this Float and Double class contains NaN constant hence if the result is undefined we won't get any arithmetic exception in floating point arithmetic.

System.out.println(0.0/0); → NaN

System.out.println(-0.0/0); → NaN

Note: *** for any x value including NaN the following expressions returns false

X < NaN;

X <= NaN;

X > NaN;

X >= NaN;

X == NaN;

For any x value including NaN the following expression returns true

X != NaN;

Example:

System.out.println(10 < Float.NaN); → false

System.out.println(10 <= Float.NaN); → false

System.out.println(10 > Float.NaN); → false

System.out.println(10 >= Float.NaN); → false

System.out.println(10 == Float.NaN); → false

System.out.println(Float.NaN == Float.NaN); → false

System.out.println(10 != Float.NaN); → true

System.out.println(Float.NaN != Float.NaN); → True

Arithmetic Exception :

1. It is runtime exception but not compile time error
2. It is possible only in integral arithmetic but not in floating point arithmetic
3. The only operators which cause arithmetic exception are ‘/’ and ‘%’

String concatenation operator (+):

The only overloaded operator in java is ‘+’ operator sometimes it acts as arithmetic addition operator and some it act as string concatenation operator.

If at least one argument is string type then + operator act as concatenation operator and if both arguments are number type then + operator act as arithmetic addition operator

Example 1:

```
String a= "malachi";
Int b=10, c=20, d=30;
System.out.println(a+b+c+d); → malachi102030
System.out.println(b+c+d+a); → 60malachi
System.out.println(b+c+a+d); → 30malachi30
System.out.println(b+a+c+d); → 10malachi2030
```

```
a+b+c+d
"malachi10"+c+d
"malachi1020"+d
"malachi102030"
```

Example 2:

Consider the following declarations?

```
String a = "malachi";
int b = 10, c = 20, d = 30;
```

which of the following expression are valid?

```
a = b + c + d; → CE: incompatible types; found: int; required: java.lang.String;
a = a + b + c; → valid
b = a + c + d; → CE: incompatible types; found: java.lang.String; required: int;
b = b + c + d; → valid
```

Relational operator (<, <=, >, >=):

We can apply relational operator for every primitive type except Boolean

Example:

```
system.out.println(10 < 20); → true
system.out.println('a' < 10); → false
system.out.println('a' < 97.6); → true
system.out.println('a' > 'A'); → true
```

system.out.println(true < false); → CE: operator > symbol cannot applied to Boolean, Boolean

We can't apply relational operators for object types

Example:

```
System.out.println("malachi123" > "malachi")
```

→ CE: operator > cannot be applied to java.lang.String, java.lang.String;

Nesting of relational operators is not allowed otherwise we will get compile time error

```
System.out.println( 10 < 20 < 30);
```

→ CE: operator < cannot be applied to Boolean,int

Equality operator (`==`, `!=`):

We can apply equality operators for every primitive including Boolean type also

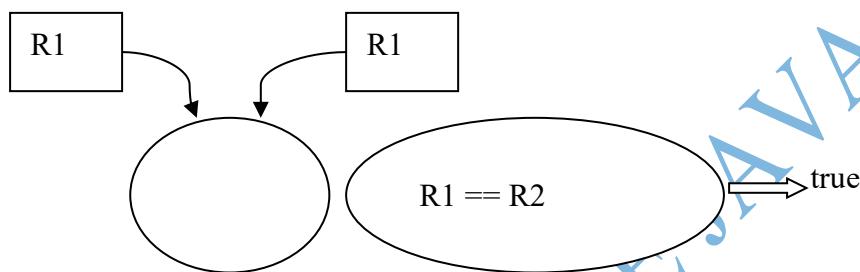
Example:

```
System.out.println(10 == 20); → false  
System.out.println('a' == 'b'); → false  
System.out.println('a' == 97.0); → true  
System.out.println(false == false); → true
```

We can apply equality operators for objects types also

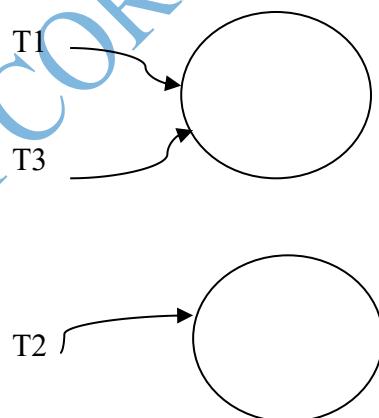
For object references R1, R2

`R1 == R2` returns true if and only if both reference pointing to the same object (reference comparison or address comparison)



Example 1:

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



If we apply equality operators for object types then compulsory there should be some relation between argument types(either child to parent or parent to child or same type) other wise we will get compile time error saying “ **incomparable types: java.lang.String and java.lang.thread**”.

Example:

```
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
```

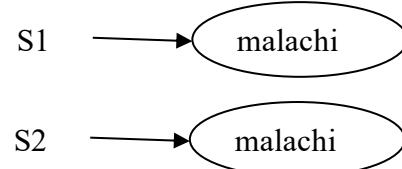
`System.out.println(s == t);` → CE: incomparable types: `java.lang.String` and `java.lang.thread`

What is the difference between == operator and .equals() method

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

Example:

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



Note: for any object reference **r**, **r == null** is always **false** but **null == null** is always **true**

Example 1:

```
String s = new String("malachi");
System.out.println (s == null);
→false
```

Example 2:

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

Example 3:

```
System.out.println (null == null);
→true
```

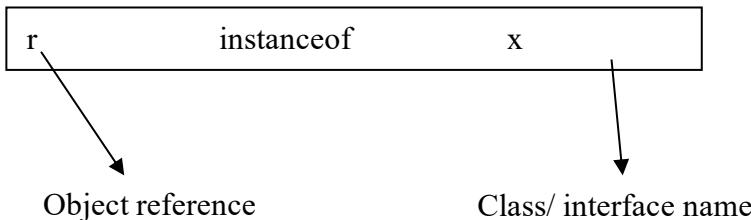
instanceof operator

we can use instanceof operator to check whether the given object is of particular type or not
example:

```
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
}
```

Syntax:

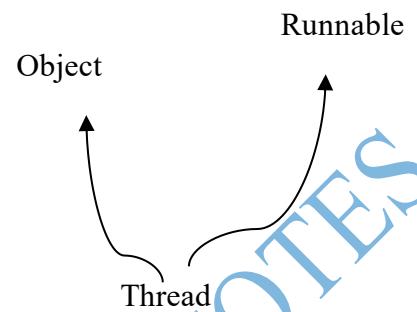
Example 1:



```

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

```



To use instanceof operator compulsory there should be some relation argument types (either child to parent or parent to child or same type) otherwise we will get compile time error saying incompatible types

Example:

```

Thread t = new Thread();
System.out.println( t instanceof String);
→ CE: incompatible type; found java.lang.Thread; required:
java.lang.String;

```

Note: for any class or interface **x**, **null instanceof x** is always false

```

System.out.println(null instanceof Thread); → false
System.out.println(null instanceof Object); → false
System.out.println( null instanceof Runnable); → false

```

Bitwise operator(&(and), |(or), ^(x-or))

& → and → returns true iff both arguments are true

| → or → return true iff atleast one argument is true

^ → x-or → returns true if and only both arguments are different

Example:

```

System.out.println(true & false); → false
System.out.println(true | false); → true
System.out.println( true ^ false); → true

```

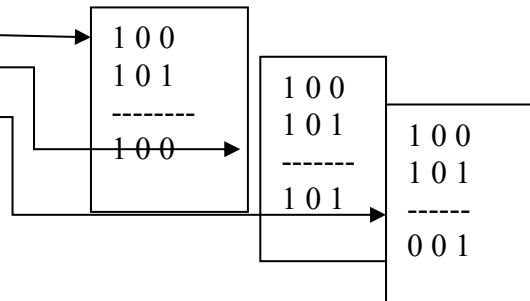
We can apply these operators for integral type also

Example:

```

System.out.println( 4 & 5); → 4
System.out.println( 4 | 5); → 5
System.out.println(4 ^ 5); → 1

```



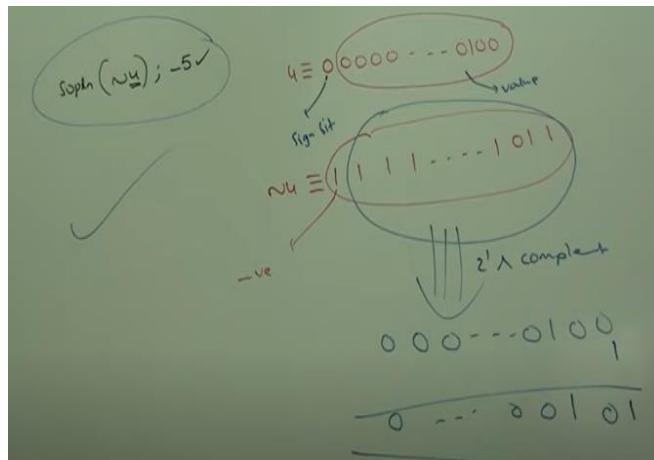
Bitwise complement operator(\sim):

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 get compile time error

Example:

System.out.println(\sim true); \rightarrow CE: operator \sim cannot be applied to Boolean

System.out.println(\sim 4); \rightarrow -5



Note: the most significant bit acts as sign bit 0 means positive number 1 means negative number. Positive numbers will be represented directly in the memory where as negative number will be represent indirectly in the memory in 2's compliment form

Boolean compliment operator (!):

We can apply this operator only for Boolean types but not for integral types

Example:

System.out.println(!4); \rightarrow CE: operator ! cannot be applied to int

System.out.println(!false); \rightarrow true

For $\&$, $|$, $^$ applicable for both Boolean and integral types

For \sim applicable for only integral types but not for Boolean type

For $!$ applicable only for Boolean but not for integral type

Short- circuit operators (**&&**, **||**):

These are exactly same as bitwise operators (**&** , **|**)except the following difference

&, 	&& ,
1. Both arguments should be evaluated always	1. Second argument evaluation is optional
2. Relatively performance low	2. Relatively performance is high
3. Applicable for both Boolean an integral types	3. Applicable only for Boolean but not for integral type

Note :

1. **X && Y :** y will be evaluated if and only if x is true that is if x is false then y won't be evaluated
2. **X || Y:** y will be evaluated if and only iff x is false that is if x is true the y won't be evaluated

Example-1:

```
int x=10,y=15;
if(++x < 10 & ++y)
{
    X++;
}
Else
{
    Y++;
}
System.out.println(x+ "----" + y);
```

	x	y
&	11	17
&&	11	16
 	12	16
 	12	16

Example 2:

```
Int x=10;
If(++x < 10 && (x/0 >10))
{
    System.out.println("hello");
}
Else
{
    System.out.println("hi");
}
```

1. Compile time error
2. Runtime Error: arithmetic expression" / by zero
3. Hello
4. Hi

Ans: hi

Note: If we replace **&&** with **&** then we will get runtime exception saying arithmetic expression / by zero

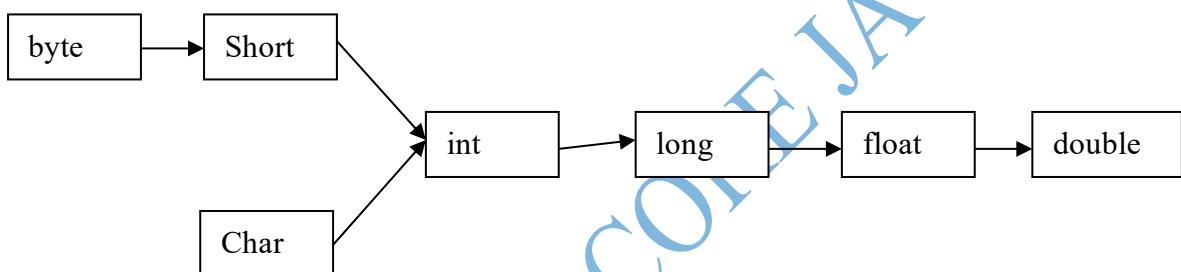
Type-cast operator:

There are two types of type casting

1. Implicit type casting
2. Explicit type casting

Implicit type casting:

1. Compiler is responsible to perform implicit type casting
2. Whenever we are assigning smaller data type value to bigger data type variable implicit type casting will be performed
3. It is also known as widening or up casting
4. There is no loss of information in this type casting
5. The following are various possible conversions where implicit type casting performed



Example -1:

Int x= 'a'; (compiler converts char to int automatically by implicit type casting)
System.out.println(x); → 97

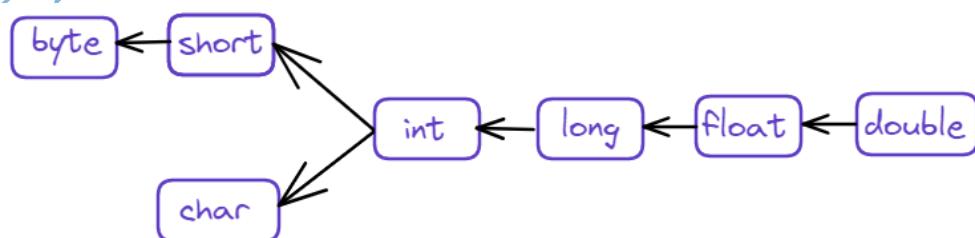
Example -2:

Double d= 10; (compiler converts int to double automatically by implicit type casting)
System.out.println(d); → 10.0

Explicit type casting:

1. Programmer is responsible to perform explicit type casting
2. When we are assingning bigger data type value to smaller data type value then explicit type casting is 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.



When we are assigning bigger data type value to smaller data type variable by explicit type casting the most significant bits will be lost we have to consider only least significant bits

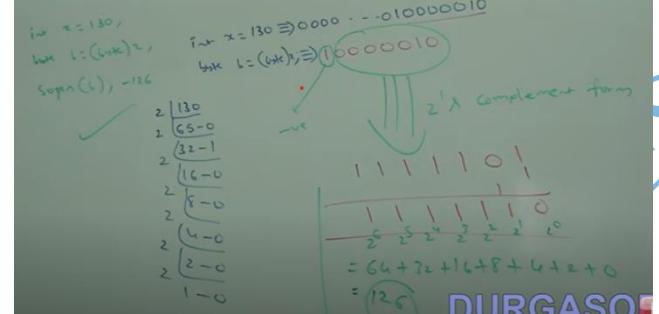
Example:

Int x= 130

Byte b= x; → CE: possible loss fo precision; found: int; required: byte;

Byte b= (byte)x;

System.out.println(b); → -126



Example – 2:

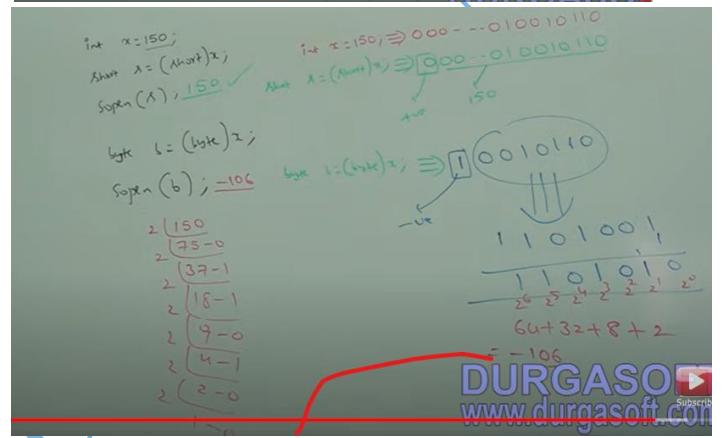
Int x = 150;

Short s= (short)x;

System.out.println(s); → 150

Byte b= (byte) x;

System.out.println(b); → -106



If we assign floating point values to the integral types by explicit type casting the digits after the decimal will be lost.

Double d=13.456

Int x=(int)d;

System.out.println(x);→130

Byte b= (byte) d;

System.out.println(b)→ 126

Assignment operators:

There are three type assignment operators

1. Simple assignment:

Example:

Int x = 10;

2. Chained assignment:

Example:

Int a,b,c,d;

A = b = c= d= 20;

System.out.println(a+ "...." +b+ " ..." +c+ "...."+d);

20 20 20 20

We can't perform chained assignment directly at the time of declaration.

Example 2:

Int a = b = c = d = 20 ;

→ CE: cannot find symbol; symbol: variable b;location : class test

Example 3:

Int b,c,d;

Int a = b = c = d = 20;

3. Compound assignment:

Some times assignmet operators mixed with some other operator such type of assignment operators are called compound operators.

Example:

Int a = 10;

A += 20;

System.out.println(a); → 30

The following are all possible compound assignment operators in java

+= &= >>=

-= |= >>>=

*= ^= <<=

/=

%=

4. In the case compound assignment operators internally type casting will be perform automatically

Example 1:

Byte b= 10;

B = b+1;

System.out.println(b);

Max(int, byte, int)

= int

→ CE: possible loss of precision; found: int required: byte

Example 2:

Byte b =10;

B++;

System.out.println(b); → 11

B = (byte)(b+1)

example3:

byte b =10;

b += 1;

system.out.println(b); → 11

b=(byte)(b+1)

Example 4:

```
Byte b = 127  
B += 3;  
System.out.println(b); → -126
```

Example 5:

```
Int a,b,c,d;  
A = b = c = d = 20;  
A += b -= c *= d /= 2;  
System.out.println(a + " --- " + b + " --- " + c + " --- " + d); → -160 --- -180 --- 200 --- 10
```

Conditional operator (? , :)

The only possible ternary operator in java is conditional operator

Syntax:

```
Int x =(10 < 20)? 30 : 40;  
System.out.println(x); → 30
```

We can perform nesting of conditional operator also.

```
Int x = )10 > 20)? 30: ((40 > 50)? 60 : 70);  
System.out.println(x); → 70
```

new operator :

we can use new operator to create object

example:

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

Note:

1. After create an object constructor will be executed to perform an 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 object is the responsibility of garbage collector.

[] operator:

We use this operator to declare and create arrays

Example:

```
Int [ ] x = new int[ 10];
```

Java Operator Precedence:

1. Unary operators
 - a. [], x++, x--
 - b. ++x, --x, ~, !
 - c. new, <type>
2. Arithmetic operators
 - a. *, /, %
 - b. +, -
3. Shift operators:
 - a. >>, >>>, <<
4. Comparison operators:
 - a. <, <=, >, >=, instanceof
5. Equality operators:
 - a. ==, !=
6. Bitwise operators:
 - a. &
 - b. ^
 - c. |
7. Short circuit operators:
 - a. &&
 - b. ||
8. Conditional operator
 - a. ?, :
9. Assignment operators
 - a. =, +=, -=, *=

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

```
class Test
{
    Public static void main(String[ ] args)
    {
        System.out.println(m1(1)+m2(2)*m1(3)/m1(4)+m1(5)*m1(6));
    }
    Public static int m1(int i)
    {
        System.out.println(i);
        return i;
    }
}
```

Output:
1
2
3
4
5
6
32

1 + 2 * 3 / 4 + 5 * 6
1 + 6 / 4 + 5 * 6
1 + 1 + 5 * 6
1 + 1 + 30
2 + 30 = 32

new vs newInstance():

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

example:

```
Test t = new Test();
Student s = new Student();
Customer c = new customer();
```

2. newInstance() is a method present in class Class we can use newInstance() method to create an object if we don't know class name at the beginning and it is available dynamically at run time.

Example:

```
class Student
{
}
Class Customer
{
}
Class Test
{
    Public static void main(String[ ] args)throws Exception
    {
        Object o = Class.forName(args[0]).newInstance();
        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

3. in the case of new operator base on our requirement we can invoke any constructor

example:

```
Test t = new Test();
Test t1 = new Test(10);
Test t2 = new Test ("malachi");
```

4. But newInstance() method internally calls no argument constructor hence to use newInstance() method compulsory corresponding class should contain no-argument constructor otherwise we will get runtime exception saying instantiation exception

Example:

```
Class Test
{
    Test()
    {
        System.out.println("no-arg constructor"); RE: instantiation exception
    }
    Public static void main(String [] args )throws Exception
    {
        Object o = Class.forName(args[0].newInstance());
        System.out.println(" object created for:"+
+o.getClass().getName());
    }
}
```

5. While using new operator at run time if the corresponding .class file is not available then we will get runtime exception saying noClassDefFoundError: test

Example:

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

At run time 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 is not available then we will get runtime exception saying ClassNotFoundException:
Test123

Example:

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

```
Java Test Test123;
```

At runtime if Test123.class file is not available then we will get runtime exception saying ClassNotFoundException: Test123

Difference between new and newInstance:

new	newInstance()
1. it is operator in java	1. it is a method present in java .lang.class
2. we can use new operator to create object if we know class name at the beginning.	2. we can use newInstance() method to create object if we don't know class name at the beginning and it is available dynamically at run time
3. to use new operator class not required to contain no-arg constructor	3. to use newInstance() method compulsory class should contain no-arg constructor otherwise we will get runtime exception saying instantiation exception
4. at runtime if class file not available then we will get runtime exception saying NoClassDefFoundError ,which is unchecked	4. at runtime if the corresponding .class file is not available then we will get runtime exception saying ClassNotFoundException, which is checked

Difference between ClassNotFoundException vs NoClassDefFoundError

For hard-coded class names, at runtime if the corresponding .class file is not available then we will get runtime exception saying NoClassDefFoundError, which is unchecked.

Example:

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

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

For dynamically provided class name at runtime if the corresponding .class file is not available then we will get runtime exception saying ClassNotFoundException , which is checked exception

Example:

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

Java Test Student

At runtime if Student.class is not available then we will get runtime exception saying ClassNotFoundException: Sudent

Difference between instanceof vs isInstance():

Instanceof is an operator in java we can use instanceof to check whether the given objet is of particular type or not and we know the type at the beginning.

Example:

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

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

example:

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

Java Test Runnable

o/p: true

java Test String

o/p: flase

isInstance () is method equivalent of instanceof operator

Flow – control:

Flow control describes the order in which the statement will be executed at runtime
Flow control are classified in three types

1. Selection statement
 - a. If – else
 - b. Switch()
2. Iterative statements
 - a. While()
 - b. Do- while()
 - c. For()
 - d. For- each loop (1.5v)
3. Transfer statements
 - a. Break
 - b. Continue
 - c. Return
 - d. Try- catch- finally
 - e. Assert(1.4v)

Selection statement:

1. if- else:

Syntax:

```
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 compile time error.

Example 1:

```
Int x = 0;
If(x) →CE: incompatible type found int, required: Boolean
{
    System.out.println("hello");
}
Else
{
    System.out.println("hi");
}
```

Example 2:

```
Int x = 10;
If(x=20) →CE: incompatible type found int, required: Boolean
{
    System.out.println("hello");
}
Else
{
    System.out.println("hi");
}
```

Example 3:

```
Int x = 10;
If(x== 20)
{
    System.out.println("hello");
}
Else
{
    System.out.println("hi");
}
```

o/p: hi

Example 4:

```
Boolean b = true;
If(b = flase)
{
    System.out.println("hello");
}
Else
{
    System.out.println("hi");
}
```

o/p: hi

Example 5:

```
Boolean b = false;
If(b == false)
{
    System.out.println("hello");
}
Else
{
    System.out.println("hi");
}
```

o/p: hello

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

example 1:

```
if (true)
    System.out.println("hello"); → valid
```

example 2:

```
if(true); → valid
```

example 3:

```
if (true)
    int x = 10; → invalid Compiletime error well get
```

example 4:

```
if(true)
{
    Int x=10; → valid
}
```

Note: semicolon(;) is a valid java statement which is also known as empty statement

Note: there is no dangling problem in java, every else is mapped to the nearest if statement

Switch ()

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.

Syntax:

```
Switch(x)
{
    Case 1:
        Action 1;
        Break;
    Case 2:
        Action 2;
        Break;
    .....
    .....
    Case N:
        Action n;
        Break;
    Default:
        Default action
}
```

- The allowed argument types for the switch statements are byte, short, char, int until 1.4 version, but 1.5 version onwards corresponding wrapper classes and enum type also allowed

From 1.7 version onwards String type also allowed

1.4 version	1.5 version	1.7 version
Byte	Byte	
Short	Short	
Char	Character	String
Int	Integer	
	enum	

- Curly braces are mandatory except switch every curly braces or optional;
- Both case and default or optional that is the empty switch statement is valid java syntax.

Example:

```
Int x=10
Switch(10
{
}
}
```

- Inside a switch every statement should be under some case or default that is independent statements are not allowed inside a switch otherwise we will get compile time error.

Example:

```
Int x=10;
Switch(x)
{
    System.out.println("hello");
    → CE: case, default , or } expected
}
```

Every case label should be constant that is constant expression

Example:

```
Int x =10;
Int y=20;
Switch(x)
{
    Case 10:
        System.out.println(10);
        Break;
    Case y: → CE: constant expression required
        System.out.println(20);
        Break;
}
```

NOTE: If we declared Y a final then we won't get any compile time error;
Both switch argument and case label can be expressions but case label should be constant expression

Example:

```
Int x=10;
Switch(x+1)
{
    Case 10:
        System.out.println(10);
        Break;
    Case 10+20+30:
        Systm.out.println(60);
}
```

Every case label should be in the range of switch argument type otherwise we will get compile time error.

Example:

```
Byte b=10;
Switch(b)
{
    Case 10:
        System.out.prinlnt(10);
        Break;
    Case 100:
        System.out.prinlnt(10);
        Break;
    Case 1000: →CE: POSSIBLE LOSS OF PRECISION
                FOUND: INT
                REQUIRED: BYTE
        System.out.prinlnt(10);
        Break;
}
```

Example :

```
Byte b=10;
Switch(b+1)
{
    Case 10:
        System.out.println(10);
        Break;
    Case 100:
        System.out.println(10);
        Break;
    Case 1000:
        System.out.println(10);
        Break;

}
```

Duplicate case labels are not allowed other wise we will get compile time error

Example:

```
Int x=10;
Switch(x)
{
    Cast 97:
        System.out.println(97);
        Break;
    Cast 98:
        System.out.println(98);
        Break;
    Cast 99:
        System.out.println(99);
        Break;
    Cast 'a': →CE: DUPLICATE CASE LABEL
        System.out.println('a');
        Break;

}
```

Case label summary:

- ➔ It should be constant expression
- ➔ The value should be in the range of switch argument type
- ➔ Duplicate case labels are not allowed

Fall-through inside switch:

Within the switch if any case is matched from that case onwards all statement will be executed until break or end of the switch. This is called fall through inside switch. The main advantage of fall-through inside a switch is we can define common action for multiple cases(code reusability).

Example:

```
Switch(x)
{
    Case 1:
    Case 2:
    Case 3:
        System.out.println("Q4");
    Case 4:
    Case 5:
    Case 6:
        System.out.println("Q4");
}
```

Example:

Output:
X = 0
0
1
X = 1
1
X = 2
2
Def
X = 3
def

```
Switch(x)
{
    Case 0:
        System.out.println(0);
    Case 1:
        System.out.println(1);
        Break;
    Case 2:
        System.out.println(2);
    Case 3:
        System.out.println("def");
}
```

Default case:

1. Within the switch we can take default cast atmost once.
2. Default case will be executed if and only if there is no case matched
3. Within the switch we can write default case anywhere but it is recommended to write as last case

Example:

Output:
X = 0
0
X = 1
1
2
X = 2
2
Def
X = 3
Def
0

```
Switch(x)
{
    Default:
        System.out.println("def");
    Case 0:
        System.out.println(0);
    Case 1:
        System.out.println(1);
    Case 2:
        system.out.println(2);
}
```

Iterative statements

while ():

if we don't know number of iteration in advance then we should go for while() loop

Example:

```
while(rs.next())
{
}
```

```
while(e.hasMoreElements())
{
}
```

```
while(itr.hasNext())
{
}
```

Syntax :

```
While(b) → should be Boolean type
{
    Action;
}
```

The argument should be Boolean type if we are trying to provide any other type then we will get compile time error.

Example:

```
While(1)
{
    System.out.println("hello");
    → CE : incompatible types;
    Found: int
    Required: boolean
}
```

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

Example 1:

```
While (true)
    System.out.println("hello");
    → Valid
```

Example 2:

```
While(true);
    → Valid
```

Example 3:

```
While(true)
    Int x = 10;
    → Invalid
```

Example 4:

```
While(true)
{
    Int x =10;
}
    → Valid
```

Example 1:

```
While(true)
{
    System.out.println("hello");
}
System.out.println("hi"); → CE :
unreachable statement
```

Example 2:

```
While(false)
{
    System.out.println("hello"); → CE :
unreachable statement {
}
System.out.println("hi");
```

Example 3:

```
Int a=10,b=20;
While(a<b)
{
    System.out.println("hello");
}
System.out.println("hi");
```

Output:

```
Hello
Hello
..... infinity
```

Example 4:

```
Int a=10,b=20;  
While(a>b)  
{  
    System.out.println("hello");  
}  
System.out.println("hi");
```

Out put:

Hi

Example 5:

```
Final Int a=10,b=20;  
While(a<b)  
{  
    System.out.println("hello");  
}  
System.out.println("hi"); → CE:  
unreachable statement
```

Example 6:

```
Int a=10,b=20;  
While(a<b)  
{  
    System.out.println("hello");  
→ CE: unreachable statement  
}  
System.out.println("hi");
```

Note:

1. Every final variable will be replaced by the value at compile time only.

Example

```
Final int a = 10;  
Int b = 20;  
System.out.println(a); ==> after compilation → sopln(10)  
System.out.println(b); ==> after compilation → sopln(b)
```

2. If every argument is a final variable (compile time constant) then that operation should be performed at compile time only

Example:

```
Final int a = 10, b = 20;  
Int c = 20;  
System.out.println(a+b); ==> after compilation → sopln(30)  
System.out.println(a+c); ==> after compilation → sopln(10+c)  
System.out.println(a < b); ==> after compilation → sopln(true)  
System.out.println(a < c); ==> after compilation → sopln(10 < c)
```

do-while ():

if we want to execute loop body at least 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 statement between do and while which should not be declarative statement.

Example:

```
Do
    System.out.println("hello");
While(true);
      → valid
```

Example 2:

```
Do;
While(true);
      → Valid
```

Example:

```
Do while(true)
System.out.println("hello");
While(false);
```

Example 3:

```
Int x = 10;
While(true);
      → Invalid
```

Example 4:

```
Do
{
    Int x =10;
}while(true);
      → Valid
```

Example 5:

```
Do
While(true);
      → Invalid
```

Example:

```
Do
    While(true)
        System.out.println("hello");
    While(false);
```

Output:

```
Hello
Hello
Hello
.....
.....
Infinity
```

Example:

```
do
{
    System.out.println("hello");
}while(true);
System.out.println("hi"); →CE: unreachable
statement
```

Example 2:

```
do
{
    System.out.println("hello");
}while(false);
System.out.println("hi");
```

Output:
Hello
Hi

Example 3:

```
Int a =10, b = 20;
do
{
    System.out.println("hello");
}while(a < b);
System.out.println("hi");
```

Output:
Hello
Hello
Hello
.....
.....
.....(infinity)

Example 4:

```
Int a = 10,b =20;
Do
{
    System.out.println("hello");
}while(a > b);
System.out.println("hi");
```

Output:
Hello
Hi

Example 5:

```
Final Int a = 10, b =20;
do
{
    System.out.println("hello");
}while(a < b);
System.out.println("hi"); →CE: unreachable statement
```

Example 6:

```
Final Int a = 10,b =20;
do
{
    System.out.println("hello");
}while(a < b);
System.out.println("hi");
```

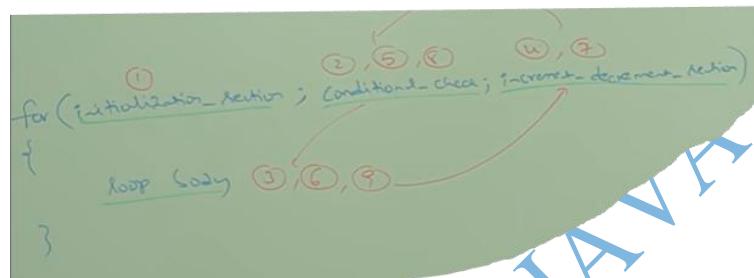
Output:
Hello
Hi

For loop

1. For loop is the most commonly used loop in java
2. If we know number of iteration in advance then for loop is the best choice.

Syntax:

```
For(initialization_section ; conditional_check, increment_decrement_section)
{
    Loop body;
}
```



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

Example:

```
For(int i= 0; true; i++)
{
    System.out.println("hello");
}
```

Example:

```
For(int i= 0; true; i++);
```

Example:

```
For(int i= 0; true; i++)
Int x= 10;
```

Initialization section:

- This part will be executed only in loop lifecycle
- Here we can declare and initialize local variables of for loop
- Here we can declare any number variables but should be of the same type by mistake if we are trying to declare different data type variables then we will get compile time error

Example:

int j = 0, j = 0; → valid
int i = 0, String s = "durga"; → invalid
int i = 0, int j = 0; → invalid

- In the initialization section we can take any valid java statement including SOP

Example:

```
Int I = 0;
For(System.out.println("hello boss u r sleeping"); i<3; i++)
{
    System.out.println("no boss u only sleeping");
}
```

Output:

```
Hello boss u r sleeping
No boss u only sleeping
No boss u only sleeping
No boss u only sleeping
```

Conditional section:

- Here we can take any valid java expression but should be of the Boolean
- This part is optional and if we are not taking anything compiler will always place true

Syntax:

```
For( int I = 0; true; i++)
{
    Body of the loop;
}
```

Increment or Decrement section:

In the increment or decrement section we can take any valid java statement including SOP

Example:

```
int i = 0;
for(System.out.println("hello"); i < 3; system.out.println("hi"))
{
    i++;
}
```

Output:

```
Hello
Hi
Hi
Hi
```

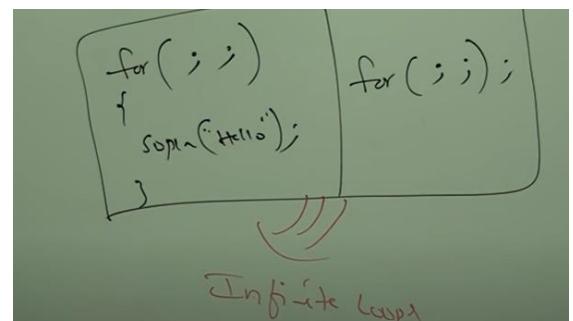
All three parts for loop are independent of each other and optional

Example:

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

Example:

For(; ;);
Infinite loop



Case 1:

```
For(int I = 0; true; i++)  
{  
    System.out.println("hello");  
}  
System.out.println("hi"); → CE: unreachable statement
```

Case 2:

```
For(int I = 0; false; i++)  
{  
    System.out.println("hello");  
}  
System.out.println("hi"); → CE: unreachable statement {
```

Case 3:

```
For(int I = 0; ; i++)  
{  
    System.out.println("hello");  
}  
System.out.println("hi"); → CE: unreachable statement
```

Case 4:

```
Int a =10, b= 20;  
For(int I = 0; a < b ; i++)  
{  
    System.out.println("hello");  
}  
System.out.println("hi");
```

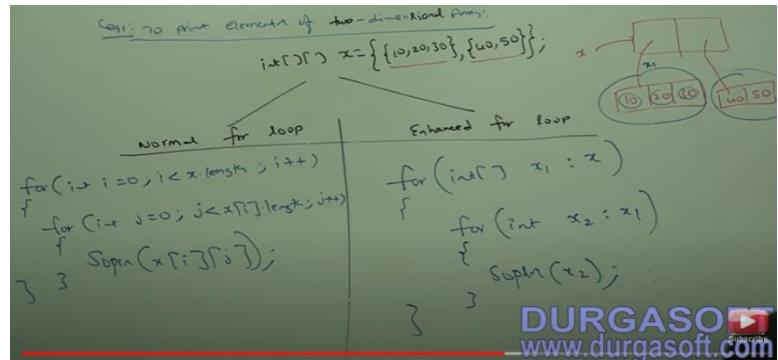
Output:
Hello
Hello
.....
.....

Case 5:

```
Int a =10, b= 20;
For(int I = 0; a>b ; i++)
{
    System.out.println("hello");
}
System.out.println("hi");
```

Output:

Hi



Case 6:

```
Final Int a =10, b= 20;
For(int I = 0; a>b ; i++)
{
    System.out.println("hello");
}
System.out.println("hi"); → CE: unreachable statement
```

Case 7:

```
Final Int a =10, b= 20;
For(int I = 0; a>b ; i++)
{
    System.out.println("hello");
}
System.out.println("hi"); → CE: unreachable statement
```

For each(enhanced for loop 1.5 v)

Introduced in 1.5 versions. It is specially designed loop to retrieve elements of arrays and collections

Example 1:

To print elements of one dimensional array

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

Normal for loop	Enhanced for loop
<pre>For(int I = 0; I < x.length; i++) { System.out.println(x[I]); }</pre>	<pre>For(int x1 : x) { System.out.println(x1); }</pre>

Example 2:

To print elements of two dimensional array

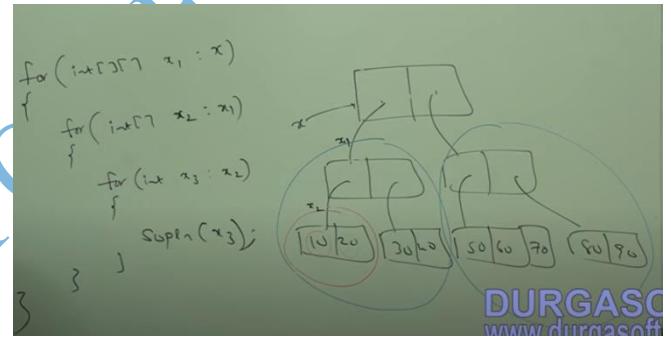
```
Int [ ] [ ] x = {{10,20,30}{40,50}}
```

Normal for loop	Enhanced for loop
<pre>For (int I = 0; i<=x.length; i++) { For(int j = 0; j<= x[i].length; j++) { System.out.println(x[i][j]); } }</pre>	<pre>For(int[] x1 : x) { For(int x2 : x1) { System.out.println(x2); } }</pre>

Example 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 retrieve elements of arrays and collections but it's a limitation is it is applicable only for arrays and collections and it is not a general purpose loop

```
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

```

Int [ ] x = {10,20,30,40,50};
For(int I = x.length - 1; I >= 0;i--)
{
    System.out.println(x[i]);
}

```

→ We can't write an equivalent for-each loop directly

Output: 50

```

40
30
20
10

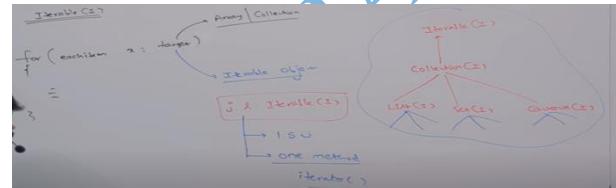
```

Iterable(I)

```

For(eachitem x : target)
{
-----
-----
}

```



- The target element in for-each loop should be iterable object
- An object is said to be iterable if and only if the corresponding class implements java.lang.Iterable(I) interface.
- Iterable interface introduced in 1.5 version
- And it contains only method iterator()

Public Iterator iterator()

- All array related classes and collection implemented classes already implement iterable interface being a programmer we are not required to do anything just we should aware the point

Difference between iterator and iterable?

Iterator(I)	Iterable(I)
1. it is related to collections	1. it is related to for-each loop
2. we can use to retrieve elements of collections one by one	2. the target element in for-each loop should be iterable
3. present in java.util package	3. present in java.lang package
4. it contains 3 methods <ul style="list-style-type: none"> • hasNext() • next() • remove() 	4. it contains one method <ul style="list-style-type: none"> • Iterator()

Transfer statements

Break:

We can use break statement in the following statement

1. inside switch to stop fall through

```
Int x= 0 ;
Switch(x)
{
    Case 0:
        System.out.println(0);
    Case 1:
        System.out.println(1);
        Break;
    Case 2 :
        System.out.println(2);
    Default:
        System.out.println("def");
}
```

Out put:

```
0
1
```

2. inside loops:

Break loop execution based on some condition

```
For(int I = 0; I < 10; i++)
{
    If (I == 5)
        Break;
    Sopl(i);
}
```

Output:

```
0
1
2
3
4
```

3. inside labelled blokes:
to break block execution base on some condition

```
class Test{  
    public static void main(String [ ] args){  
        int x = 10;  
        11:  
        {  
            System.out.println("begin");  
            If(x ==10;  
                Break 11;  
            System.out.println("end");  
        }  
        System.out.println("hello");  
    }  
}  
Output:  
Begin  
Hello
```

These are the only places where we can use break statement, if we are using anywhere else we will get compile time error saying break outside switch or loop

```
Class Test  
{  
    Public static void main(String[ ] arg)  
    {  
        Int x = 10;  
        If(x ==10); -> CE: break outside switch or loop  
        Break;  
        System.out.println("hello");  
    }  
}
```

Continue:

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

```
For(int I= 0; I < 10; i++)  
{  
    If(I % 2 == 0)  
        Contine;  
    System.out.println(i);  
}  
Output:  
1  
3  
5  
7  
9
```

We can use continue statement only inside loops if we are using anywhere else we will get compile time error saying continue outside of loop

```
Class Test
{
    Public static void main(String [ ] args)
    {
        Int x= 10;
        If( x == 10 ) → continue out side of loop;
        Continue;
        System.out.println("hello");
    }
}
```

Labelled break and continue:

We can use labelled break and continue to break or continue the particular loop in nested loops

```
L1:
For( )
{
    For()
    {
        For()
        {
            Break l1;
            Break l2;
            Break l3;
        }
    }
}
```

Example:

```
L1:
For(int I= 0; I < j; i++)
{
    For(int j=0; j< 3; j++)
    {
        If(I ==j)
            Continue l1;
        System.out.println(i+”----“+j);
    }
}
```

Output

Break;

```
1 ---- 0  
2 ---- 0  
2 ---- 1
```

Break 11:

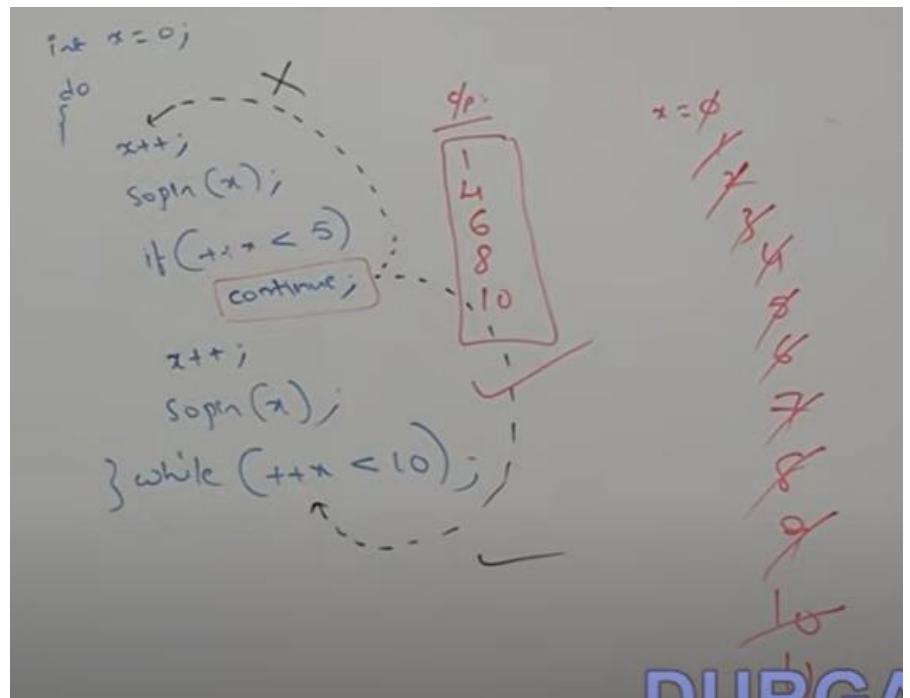
No output

Continue:

```
0 ---- 1  
0 ---- 2  
1 ---- 0  
1 ---- 2  
2 ---- 0  
2 ---- 2
```

Continue 11:

```
1 ---- 0  
2 ---- 0  
2 ---- 1
```



Do-while v/s continue (dangerous combinations)

Example:

```
Int x = 0;  
D0  
{  
    X++;  
    System.out.println(x);  
    If(++x < 5)  
        Continue;  
    X++;  
    System.out.println(x);  
}while(++x < 10);
```

Output:

```
1  
4  
6  
8  
10
```

Declarations and access modifiers

1. Java source file structure
2. Class level modifiers
3. Member level modifiers
4. Interfaces

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 compile time error.

Example

```
Class A
{
}
Class B
{
}
Class c
{
}
```

Case 1:

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

Example

```
a.java
b.java
c.java
durga.java
```

case 2:

if class B is public then name of the program should be B.java otherwise we will get compile time error saying class B is public, should be declared in a file named B.java

case 3:

if class B and C declared as public and name of the program is B.java then we will get compile time error saying class C is public, should be declared in file named C.java.

Example:

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

```

class B
{
    public static void main(String[] args)
    {
        System.out.println("B-Hello World!");
    }
}
class C
{
    public static void main(String[] args)
    {
        System.out.println("c-Hello World!");
    }
}
class D
{
}

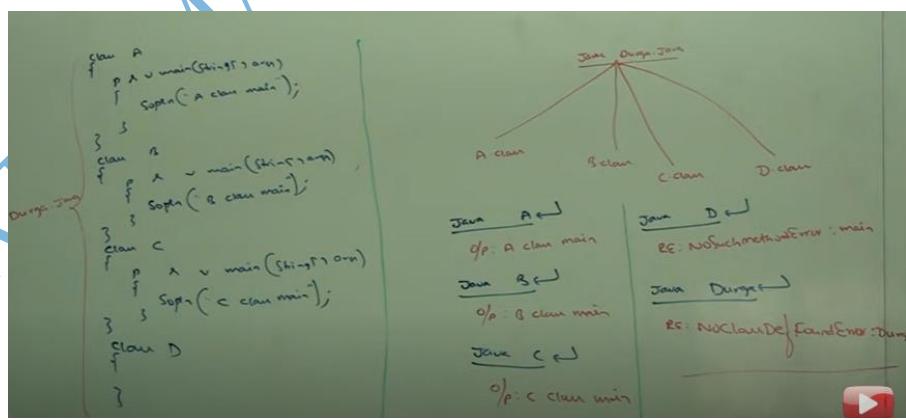
```

Out put:

```

Java A
A-Hello World!
Java B
B-Hello World!
Java C
C-Hello World!
Java D
RE: nosuchmethod error main
Java Malachi
RE: noclassdeffounderror: Malachi

```



Conclusions :-

1. Whenever we are compiling a java program for every class present in the program a separate (dot class).class file will be generated.
2. We can compile a java program (java source file) but we can run a java .class
3. Whenever we are executing a java class the corresponding class main method will be executed
4. If the class doesn't contain main method then we will get runtime exception saying **no such method error: main**
5. If the corresponding .class file not available then we will get runtime exception saying noclassdeffound error: name.
6. It is not recommended to declare multiple classes in a single source file. It is highly recommended only one class per source file and name of the program we have to keep same as class name. the main advantage of this approach is readability and maintainability of the code will be improved.

Import statement

Class Test

```
{  
    Public static void main(String[ ] args)  
    {  
        ArrayList l = new ArrayList(); → cannot find symbol; symbol : class arraylis  
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 fully qualified name every time is it increases length of the code and reduces readability

We can solve this problem by using import statement

Whenever we are writing import statement it is not required to use fully qualified name every time we can use short name directly

```
Import java.util.ArrayList;  
Class Test  
{  
    Public static void main(String[ ] args)  
    {  
        ArrayList l = new ArrayList(); → short name  
    }  
}
```

Hence import statement act as typing short cut

Case 1: types of import statements

There are two types of import statements

1. explicit class import
2. implicit class import

Explicit class import:

Example:

```
import java.util.ArrayList ;
```

it is highly recommended to use explicit class import because it improves readability of the code

best suitable for Hi-Tech city where readability is important

implicit class import:

Example:

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

not recommended to use because it reduces readability of the code

best suitable for ammerpet where typing is important

case 2:

which of the following import statement are meaning full?

Import java.util.ArrayList; → valid

Import java.util.ArrayList.*; → not valid

Import java.util.*; → valid

Import java.util; → not valid

Case 3:

Consider the following code

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

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

Note:

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

Case 4:

```
import java.util.*;
import java.sql.*;
class Test3
{
    public static void main(String[] args)
    {
        Date d = new Date(); → CE: reference to date is ambiguous
    }
}
```

Note:

Even in the case list also we may get same ambiguity problem because it is available in both util and awt packages

Case 5:

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

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

Example:

```
Import java.util.date;
Import java.util.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 interfaces present in that package by default available but not sub package classes if we want to use sub package class compulsory we should write import statement until sub package level.

Example

Java

Util

Regex

Pattern



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

Ans: 3

Case 7:

All class 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 compile time related concept if more number of imports then more will be the compile time but there is no affect on execution time(run time)

Case 9:

Difference between C-language #include and Java language import statement:

In the case C-language #include all input output header files will be loaded at beginning only (at translation time) hence it is static include

But in the case of java import statement no .class file be loaded at the beginning whenever we are using a particular class then only corresponding .class will be loaded this is consider as dynamic include or load on demand or load on fly

Note : 1.5 version new features

1. for – each loop
2. var- arg methods
3. autoboxing and autounboxing
4. generics
5. co-varient return types
6. queue
7. annotations
8. enum
9. static import

Static import

Introduced in 1.5 version, according to sun usage of static import reduces length of the code and improves readability but according to worldwide programming 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

Example

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.sqrt;
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());
    }
}
```

Explain about system.out.println():

```
Class Test
{
    Static String s = "java"
}
```

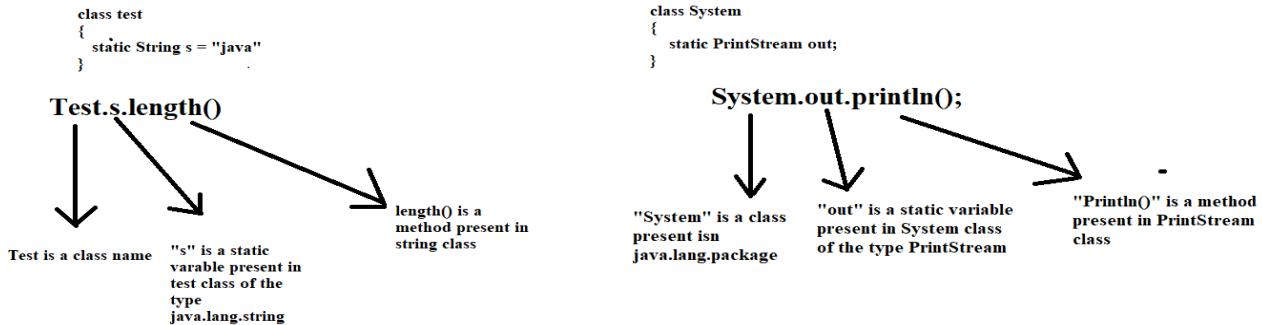
Test. S.length();

Test: “test” is class name
S: “s” is a static variable present in “test class of the type java.lang.string
Length(): “length()” is a method present in String class

```
Class System
{
    Static PrintStream out;
}
```

System.out.println();

System: “System” is class present in java.lang package
Out: “out” is a static variable present in system class of the type PrintStream
Println(): “println()” is a method present in PrintStream class



Out is a static variable present in system class hence we can access by using class name System but whenever we are writing static import it is not required to use class name and we can access out directly

Example:

```
import static java.lang.System.out;
class SystemSample
{
    public static void main(String[] args)
    {
        out.println("hello");
        out.println("hii");
    }
}
```

Example:

```
Import static java.lang.Integer.*;
Import static java.lang.Byte.*;
Public class Test
{
    Public static viod main(String[ ] args)
    {
        System.out.println(MAX_VALUE);
    }
}
```

CE: reference to MAX_VALUE is ambiguous

While resolving static members compiler will always consider the precedence in the following order

1. current class static members
2. explicit static import
3. implicit static import

Example:

```
import static java.lang.Integer.MAX_VALUE; →line 2
import static java.lang.Byte.*;
class ImportStatic
{
    static int MAX_VALUE=999; → line 1
    public static void main(String[] args)
    {
        System.out.println(MAX_VALUE);
    }
}
Output: 999
```

If we comment line 1 then explicit static import consider and hence integer class MAX_VALUE will be consider in this case the output is 2147483647

If we comment both lines 1 & 2 then implicit static import will be considered in this case output is 127(byte class MAX_VALUE)

Normal import:

1. Explicit import

Syntax:

```
Import packagename.classname;
```

Example:

```
Import java.util.ArrayList;
```

2. Implicit import

Syntax:

```
Import packagename.*;
```

Example:

```
Import java.util.*;
```

Static import:

1. Explicit static import

Syntax:

```
Import static packagename.classname.staticmember;
```

Example

```
Import static java.lang.Math.sqrt;
```

```
Import static java.lang.System.out;
```

2. Implicit static import

Syntax:

```
Import static packagename.classname.*;
```

Example:

```
Import static java.lang.Math.*;
```

```
Import static java.lang.System.*;
```

Which of the following import statements are valid?

- | | |
|---|-----------|
| 1. Import java.lang.Math.*; | → invalid |
| 2. Import Static java.lang.Math.*; | → valid |
| 3. Import java.lang.Math.sqrt; | → invalid |
| 4. Import static java.lang.Math.sqrt(); | → invalid |
| 5. Import java lang.Math.sqrt.*; | → invalid |
| 6. Import static java.lang.Math.sqrt; | → valid |
| 7. Import java.lang; | → invalid |
| 8. Import static java.lang; | → invalid |
| 9. Import java.lang.*; | → valid |
| 10. Import static java.lang.*; | → invalid |

Two packages contains a class or interface with the same name is very rare and hence ambiguity is also very rare in normal import

But two class or interfaces contains 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 between normal import and static import?

We can use normal import to import classes and interfaces of a particular package. Whenever we are writing normal import it is not required to use fully qualified name and we can 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.

Package statement

It is encapsulation mechanism to group related classes and interfaces into a single unit, which is nothing but **package**

Example 1:

All classes and interface which are required for data base operations or grouped into a single package which is nothing but **java.sql** package

Example 2:

All class and interfaces which are useful for file IO operations or grouped into a separate package which is nothing but **java.io** package

The main advantages of package are

1. To resolve naming conflicts that is unique identification of our components
2. It improves modularity of the application
3. It improves maintainability of the application
4. It provides the security of our components

There is one universal accepted naming convention for packages that is to use internet domain name in reverse

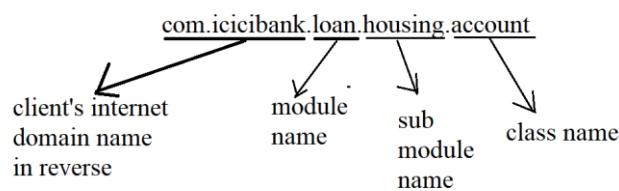
Com.icicibank.loan.housing.account

Com.icicibank : Client's internet domain name in reverse

loan : Module name

housing : Sub module name

account: Class name



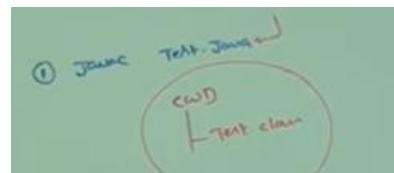
Example:

```
Package com.durgsoft.scjp;
Public class Test
{
    Public static void main(String[ ] arg)
    {
        System.out.println("package demo");
    }
}
```

Javac Test.java

Generated .class file will be placed in current working directory

Cwd
Test.class



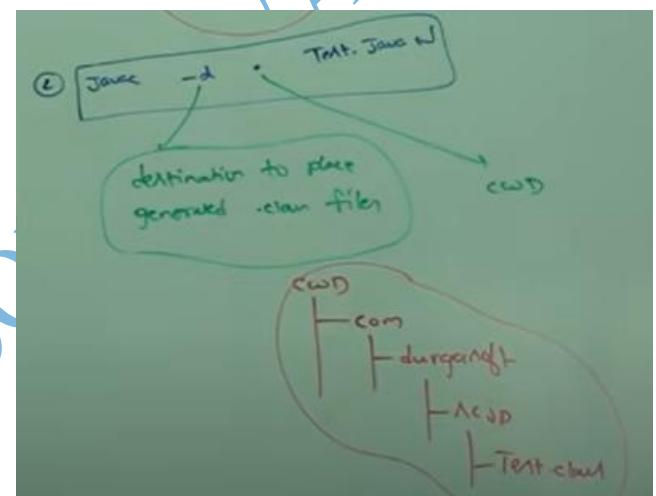
Javac -d . Test.java

-d : destination to place generated .class files

. : current working directory

Generated .class file will be placed in corresponding package structure

Cwd
Com
Durgsoft
Scjp
Test.class



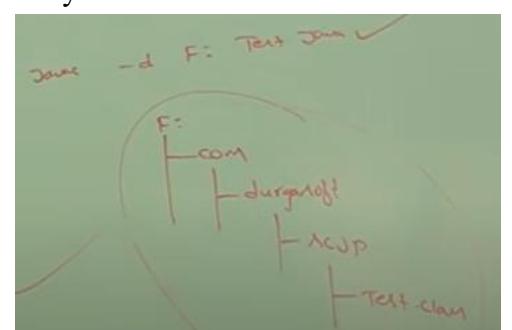
If the corresponding package structure is not already available then this command itself will create corresponding package structure.

As destination instead of dot(.) we can take any valid directory name

Example:

Javac -d F: Test.java

F
Com
Durgsoft
Scjp
Test.class



If the specified directory is not already available then we will get compile time error

Example:

Javac -d z: Test.java

If z: is not available then we will get compile time error saying directory not found: Z:
At the time of execution we have to use fully qualified name

Java com.durgasoft.scjp.Test
Output:
Package demo

Conclusion 1:

In any java source file there can be at most one package statement that is more than one package statement is not allowed otherwise we will get compile time error

Example:

```
Package pack1;  
Package pack2;  
Public class A  
{  
}
```

CE: class, interface or enum expected ;

Conclusion 2:

In any java program the first non comment statement should be package statement (if it is available) otherwise we will get compile time error.

Example:

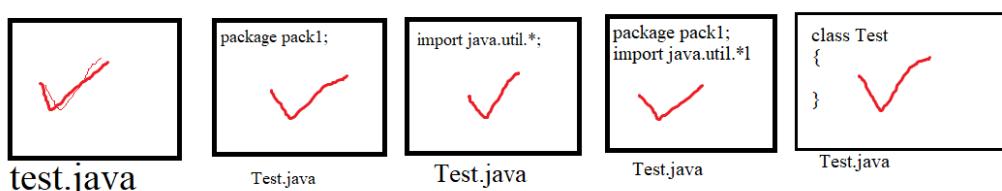
```
Import java.util.*;  
Package pack1;  
Public class A  
{  
}  
CE: class, interface or enum expected
```

The following is valid java source file structure

Package statement;	→ atmost one
Import statement;	→ any number
Class/ interface/ enum declaration	→ any number

This order is important

Note: An empty source file is valid java program hence the following are valid java source files



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 anywhere or not
2. Whether child class creation is possible or not
3. Whether the object creation is possible or not etc..

We can specify this information by using appropriate modifier

The only applicable modifiers for top level class are

```
public  
<default>  
final  
abstract  
strictfp
```

but for inner class the applicable modifiers are

```
public  
<default>  
final  
abstract  
strictfp  
+  
private  
protected  
static
```

example:

```
private class Test  
{  
    Public static void main(String[ ] args)  
    {  
        System.out.println("hello")  
    }  
}
```

CE: modifier private not allowed here

Access specifiers v/s access modifiers

Public, private, protected and default are considered as specifiers except these remaining are considered as modifiers

But this rule is applicable only for old languages like C and C++ but not for java

In java all are considered as modifiers only there is no word like specifier

Private class Test

{

}

CE: **modifier private** not allowed here

public classes

If a class declared as the public then we can access that class from anywhere

Example:

```
Package pack1;
Public class A
{
    Public void m1()
    {
        System.out.println("hello");
    }
}
```

Javac -d . A.java

Example :

```
Package pack2;
Import pack1.A;
Public class B
{
    Public static void main(String[ ] args)
    {
        A a = new A();
        a.m1();
    }
}
```

Javac -d . B.java
Java pack2.B

Output: hello

If class A is not public then while compiling B class we will get compile time error saying pack1.A is not public in pack1; cannot be accessed from outside from outside package

Default classes

If a class declared as default then we can access that class only within the current package that is from outside package we can't access hence default access is also known package level access.

Final modifier

Final is the modifier applicable for classes, methods and variables.

Final method:

Whatever method 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 it's implementation is final

Example:

```
Class P
{
    Public void property()
    {
        System.out.println("cash + land+glod");
    }
    Public final void marry()
    {
        System.out.println("subalaxmi");
    }
}
Class C extends P
{
    Public void marry()
    {
        System.out.println ("ishanayanthara");
    }
}
```

→ CE: marry() in C cannot override marry() in P; overridden method in final

Final class:

If a class declared as final we can't extend functionality of that class that is we can't create child class for that class that is inheritance is not possible for final classes

Example:

```
Final class P
```

```
{
```

```
}
```

```
Class C extends P
```

```
{
```

```
}
```

CE: cannot inherit from final P

** Note: every method present inside final class is always final by default but every variable present in final class need not be a final

Example:

```
Final class P
```

```
{
```

```
    Static int x =10;
```

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

```
{
```

```
        X=777;
```

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

```
}
```

```
}
```

The main advantage of the final keyword is we can achieve security and we can provide unique implementation

The main disadvantage of final keyword is we are missing key benefits of oops : inheritance(because of final classes) and polymorphism(because of final method s) 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, methods but not for variables

Abstract method:

Even though we don't know about implementation still we can declare a method with abstract modifier that is for abstract methods only declaration is available but not implementation hence abstract method declaration should end with semicolon(;)

Example:

```
Public abstract void m1(); → valid  
Public abstract void m1(){ } → invalid
```

Child class is responsible to provide implementation for parent class abstract methods

Example:

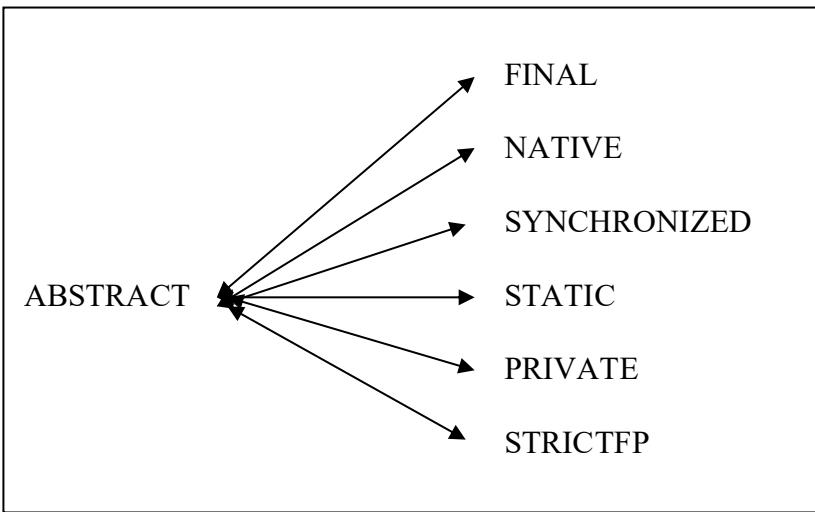
```
abstract class vehicle  
{  
    Public abstract int getNoofwheels();  
}  
  
class Bus extends Vehicle  
{  
    Public int getNofwheels()  
    {  
        Return7;  
    }  
}  
class auto extends Vehicle  
{  
    Public int getNofwheels()  
    {  
        return 3;  
    }  
}
```

By declaring abstract method in the parent class we can provide guidelines to the child classes such that which method compulsory child has to implement

Abstract method never talks about implementation if any modifier talks about implementation then it form illegal combination with abstract modifier

The following are various illegal combination of modifiers for methods with respect to abstract

Example



Abstract final void m1();

CE: illegal combination of modifiers: abstract and final

Abstract class

For any java class if we not allowed create an object (because of partial implementation) such type of class we have to declared with abstract modifier that is for abstract classes instantiation is not possible

Example:

```
Abstract class Test
{
    Public static void main(String[ ] args)
    {
        Test t = new Test();
    }
}
```

CE: test is abstract; cannot be instantiated ;

abstract class v/s abstract method

If a class contains at least one abstract method then compulsory we should declare class as abstract otherwise we will get compile time error

Reason: if a class contains 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

** even though class doesn't contain any abstract method still we can declare class as abstract if we don't want instantiation that is abstract class can contain zero number of abstract methods also

Conclusion 1:

Http servlet class is abstract but it doesn't contain any abstract methods

Conclusion 2:

Every adapter class is recommended to declare as abstract but it doesn't contain any abstract method

Example 1:

```
Class P
{
    Public void m1(); → CE: missing method body, or declare abstract
}
```

Example 2:

```
Class P
{
    Public abstract void m1();{} → CE: abstract methods cannot have a body
}
```

Example 3:

```
Class P
{
    Public abstract void m1();
} → CE: P is not abstract and does not override abstract method m1() in P
```

** If we are extending abstract class then for each and every abstract method of parent class we should provide implementation otherwise we have to declare child class as abstract in this case next level child class is responsible to provide implementation.

Example 4:

```
Abstract class P
{
    Public abstract void m1();
    Public abstract void m2();
}
Class C extends P
{
    Public void m1(){}
}
} → CE: C is not abstract and does not override abstract method m2() in p
```

final v/s abstract

1. abstract methods compulsory we should override in child classes to provide implementation were as we can't override final methods hence final abstract combination is illegal combinations for methods.
2. For final classes we can't create child class where as for abstract class we should create child class to provide implementation hence final abstract combination is illegal for class
3. ** abstract class can contain final method where as final class can't contain abstract method

Example:

Abstract class Test

```
{  
    Public final void m1 ()  
}  
}  
} → valid
```

final class Test

```
{  
    Public abstract void m1 ()  
}  
}  
} → invalid
```

Note: it is highly recommended to use abstract modifier because it promotes several oops features like inheritance and polymorphism

strictfp modifier(strict floating point)

1. introduced in 1.2 version
2. we can use strictfp for class and methods but not for variables.
3. Usually the result of floating point arithmetic is varied from platform to platform if we want platform independent result for floating point arithmetic then we should go for strictfp modifier.

Strictfp method:

If a method declared as strictfp all floating points calculations in that method has to follow IEEE 754 standard so that we will get platform independent results.

Abstract modifier never talks about implementation where as strictfp method 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 concrete method has to follow IEEE754 standard so that we will get platform independent result

We can declare abstract strictfp combination for classes that is abstract strictfp combination is legal for classes but illegal for methods.

Example 1:

```
Abstract strictfp class Test  
{  
    } → valid
```

Example 2:

```
Abstract strictfp void m1(); → CE: illegal combination of modifiers: abstract and strictfp
```

Member or variable level modifiers

Public members:

If a member declared as public then we can access that member from any where but corresponding class should be visible that is before checking member visibility we have to check class visibility.

Example:

```
Package pack1;  
Class A  
{  
    Public void m1()  
    {  
        System.out.println("A class method ");  
    }  
}
```

Javac -d . A.java → valid

```
Pakage pack2;  
Import pack1.A;  
Class B  
{  
    Public static void main(String[ ] args)  
    {  
        A a = new A();  
        a.m1();  
    }  
}
```

} Javac -d . B.java

→ invalid CE: pack1.A is not public in pack1; cannot be accessed from outside package

In the above example even though m1() is public we can't access from outside package because corresponding class A is not public that is if both class and method are public then only we can access the method from outside package

Default members:

If a member declared as default then we can access that member only with in the current package that is from out side of the package we can not access hence default access is also known package level access.

Private members:

If a member is private then we can access that member only within the class that is from outside of the class we cannot 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.

Protected members (the most misunderstood modifier in java):

If a member declared as the protected then we can access that member anywhere within the current package but only in child classes of outside package

Protected = <default> + kids

We can access protected member with in the current package anywhere either by using parent reference are by using child reference

But we can access protected member in outside package only in child classes and we should use child reference only that is parent reference cannot be used to access protected members from outside package

Example:

```
Package pack1;
Public class A
{
    Protected void m1()
    {
        System.out.println("the most misunderstood method modifier");
    }
}
Class B extends A
{
    Public static void main(String[ ] args)
    {
        1. A a = new A ();
           a.m1 ();
        2. B b = new B ();
           b.m1 ();
        3. A a1 = new B ();
           a1.m1 ();
    }
}
```

Example:

```
Package pack2;
Public class C extends A
{
    Public static void main(String[ ] args)
    {
        1. A a = new A ();
           a.m1 ();
        2. C c = new C ();
           c.m1 ();
        3. A a1 = new C ();
           a1.m1 ();
    }
}
```

→ CE: m1 () has protected access in pack1.A

We can access protected members from outside package only in child class 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

Example:

```
Pacakage 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( );
        → invalid CE
        2. C c = new C();
           c.m1( );
        → invalid Ce
        3. D d = new D();
           d.m1( );
        → valid
        4. A a1 = new C( );
           a1.m1( );
        → invalid CE
        5. A a1 = new D( );
           a1.m1( );
        → invalid CE
        6. C c1 = new D( );
           c1.m1( );
        → invalid CE
    }
}
```

→ CE: m1() has protected access is pack1.A

Summary table of private, default, protected and public modifiers:

Visibility	Private	<default>	Protected	public
Within the same class	Yes	Yes	Yes	Yes
From child class of same package	No	Yes	Yes	Yes
From non-child class of same package	No	Yes	Yes	Yes
From child class of outside package	No	No	Yes (we should use child reference only)	Yes
From non-child class of outside package	No	No	No	yes

The most restricted access modifier is private
The most accessible modifier is public

Private < default < protected < public

Recommended modifier for data member (variable) is private, but recommended modifier for methods is public.

Final variables

Final instance variables:

If the value of a variable is varied from object to object such type of variable 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 always provide default values.

Example:

```
Class Test
{
    Int x;
    Public static 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 an JVM won't provide default values

Example:

```
Class Test{
    Final int x;
}
```

CE: variable x might not have been initialized

Rule:

For final instance variables compulsory we should perform initialization before constructor completion that is the following are various places for initialization

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 initialize anywhere else then we will get compile time error.

Example:

```
Class Test
{
    Final int x;
    Publilc void m1()
    {
        X = 10;
    }
}
```

CE: cannot assign a value to final variable x

Final static variables:

If the value of a variable is not varied from object to object such type of variables are not recommended to declare as instance variables. We have to declare thoughts variables at class level by using static modifier

In the case of instance of variables for every object a separate copy will be created but in the case of static variables a single copy will be created ate class level and shared by every object of that class.

For static variables it is no requires to perform initialization explicitly JVM will always provide default values.

```

Class Test
{
    Static int x;
    Public static void main(String[ ] args)
    {
        System.out.pirntl(x); → 0
    }
}

```

If the static variable declared as final then compulsory we should perform initialization explicitly otherwise we will get compile time error and JVM won't provide any default values

```

Class Test
{
    Final static int x; → CE: variable x might not have been initialized
}

```

Rule:

For final static variables compulsory we should perform initialization before class loading completion that is 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
{
    Final static int x;
    Static
    {
        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 compile time error

```

Class Test{
    Final static int x;
    Public void m1(){
        X= 10; → CE: cannot assign a value to final variable x;
    }
}

```

final local variables:

Some times to meet temporary requirement of the programmer we have to declare variables inside a method | block | constructor such type of variables are called local variables | temporary variable | stack variable | automatic variables

For local variables JVM won't provide any default values compulsory we should perform initialization explicitly before using that local variable that is if we are not using then it is not required to perform initialization for local variable

```
Class Test
{
    Public static void main(String[ ] args)
    {
        Int x;
        System.out.println("hello"); → hello
    }
}
```

```
Class Test
{
    Public static void main(String[ ] args)
    {
        Int x;
        System.out.println(x);
    }
}
```

CE: variable x might not have been initialized

Even though local variable is final before using only we have to perform initialization that is if we are not using then it is not required to perform initialization even though it is final.

```
Class Test
{
    Public static void main(String[ ] args)
    {
        Final int x;
        System.out.println("hello"); → hello
    }
}
```

```
Class Test{
    Int x;
    Static int y;
    Public static void main (String [ ] args)
    {
        Int z = 30;
    }
}
```

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

```
{  
    Public static void main(String[ ] args)  
    {  
        Public int x =10;      → CE: illegal start of expression  
        Private Int x =10;    → CE: illegal start of expression  
        Protected int x =10; → CE: illegal start of expression  
        Static int x =10;    → CE: illegal start of expression  
        Transient int x =10; → CE: illegal start of expression  
        Volatile int x =10;  → CE: illegal start of expression  
  
        Final int x =10;     → valid  
    }  
}
```

Note:

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

Formal parameters

Formal parameters of a method simply act as local variables of that method hence formal parameters can be declared as final

If formal parameters declared as final then with in the method we can't perform reassignment

Class Test

```
{  
    Public static void main(String[ ] args)  
    {  
        M1(10,20); // actual parameters  
    }  
    Public static void m1(final int x, int y) // formal parameters  
    {  
        //x=100;      → CE: cannot assign a value to final variable x  
        Y = 200;  
        System.out.println(x"....." + y);  
    }  
}
```

Static modifier

Static is the modifier applicable for methods and variables but not for classes

We can't declare top level class with static modifier but we can declare inner class as static (such type 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.

Example:

```
class StaticVariable
{
    static int x =10;
    int y = 20;
    public static void main(String[] args)
    {
        StaticVariable sv = new StaticVariable();
        sv.x = 888;
        sv.y = 999;
        StaticVariable sv1 = new StaticVariable();
        System.out.println(sv1.x+"----"+sv1.y);
    }
}
```

output: 888 ----20

We can't access instance members directly from static area but we can access from instance area directly.

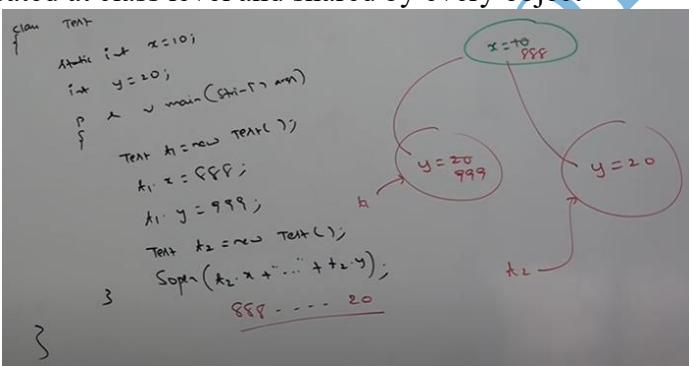
We can access static members both instant and static areas directly

Consider the following declarations

1. Int x =10;
2. Static int x = 10;
3. Public void m1()
{
 System.out.println(x);
}
4. Public static void m1()
{
 System.out.println(x);
}

Within the same class which of the above declarations we can take simultaneously

- a) I & II → valid
- b) I & IV → CE: non static variable x cannot be referenced from a static context
- c) II & III → valid
- d) II & IV → valid
- e) I & II → CE: variable x is already defined in StaticVariable
- f) III & IV → CE: m1() is already defined in StaticVariable



Case 1:

Over loading concept applicable for static methods including main method but JVM can always call string [] args main method only

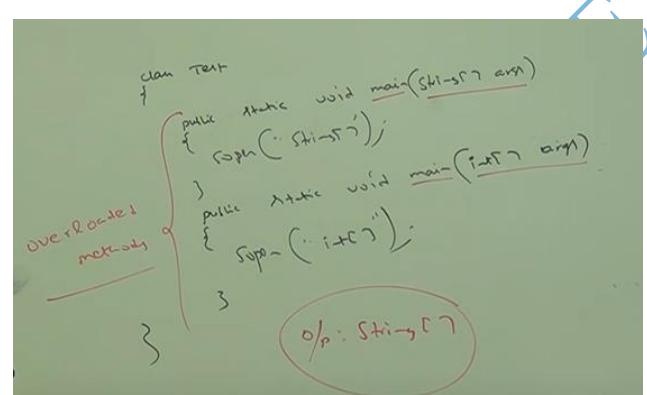
Example:

Class Test

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

Output: String []

Other overloaded we have to call just like an normal method call



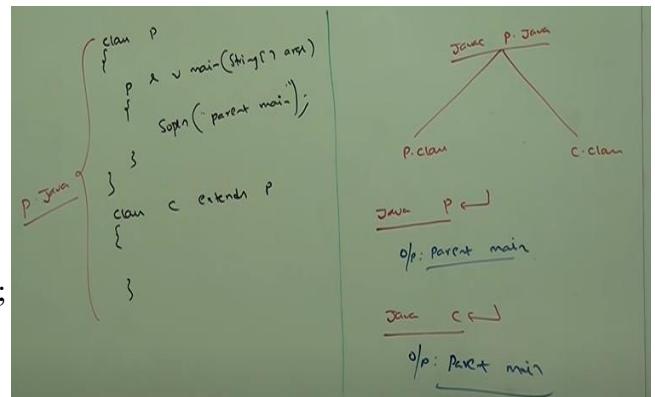
Case 2:

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

Example:

Class P

```
{  
    Public static void main(String[ ] args)  
    {  
        System.out.println("parent main");  
    }  
}  
Class C extends P  
{  
}
```



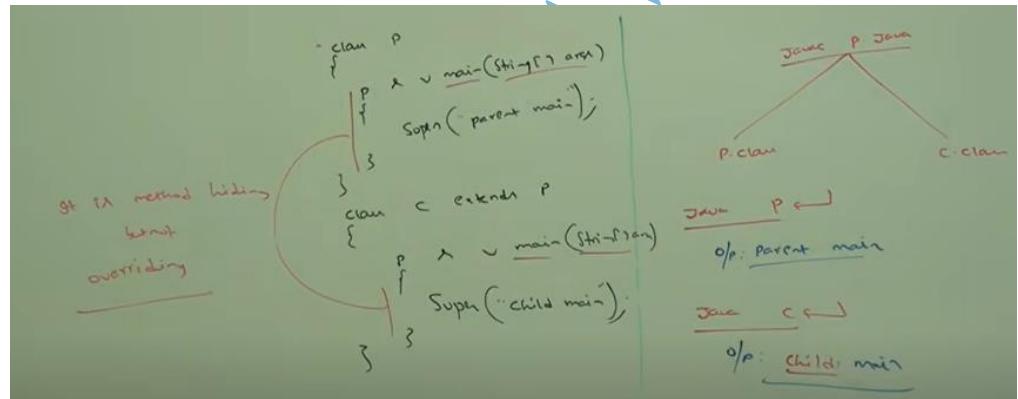
Case 3:

```
Class P
{
    Public static void main(String[ ] args)
    {
        System.out.println("parent main");
    }
}
```

→ it is method hiding but not overriding

```
Class C extends P
{
    Public static void main(String[ ] args)
    {
        System.out.println("child Main");
    }
}
```

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 static methods but it is not overriding and it is method hiding.

Note:

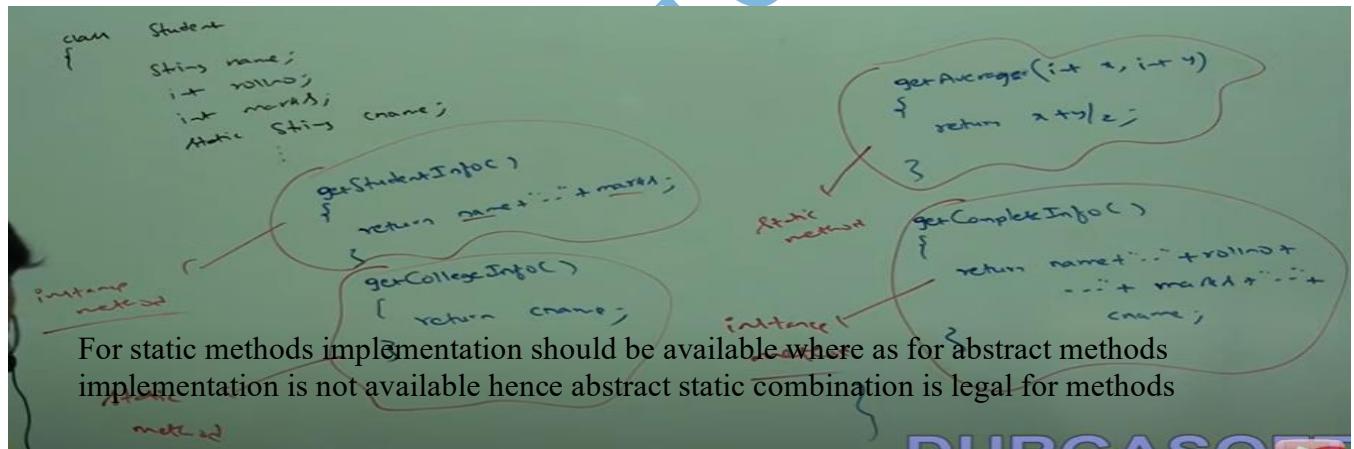
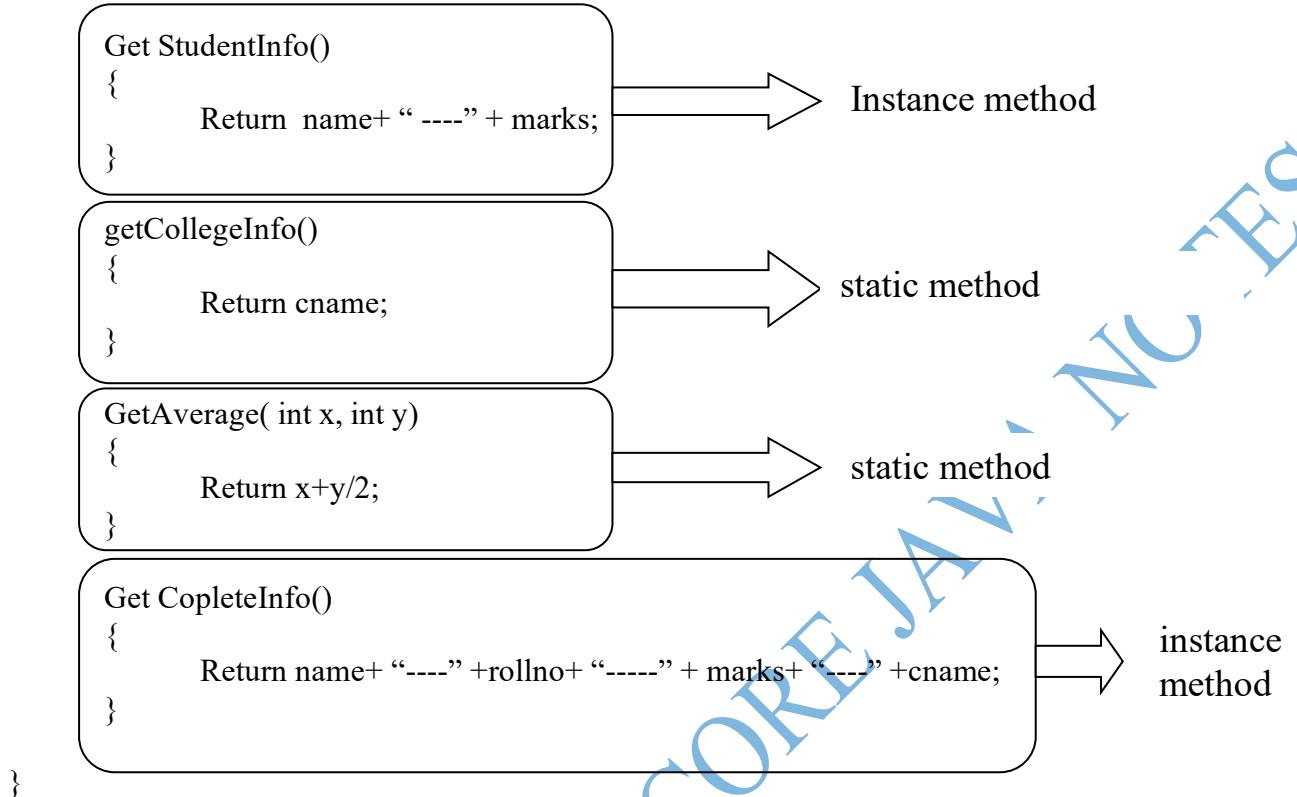
For static methods overloading and inheritance concepts are applicable but overriding concepts are not applicable but instead of overriding method hiding concept is applicable.

Inside method implementation if we are using at least 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.

Example:

```
Class Student
{
    String name;
    Int rollno;
    Int marks;
    Static String cname;
```



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 same java object then there may be 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 increase waiting time of threads and creates performance problems hence if there is no specific requirements then it not recommended to use synchronized keyword.

Synchronized method should compulsory contain implementation where as abstract method doesn't contain any implementation hence abstract synchronized is illegal combination of modifiers for methods.

native modifier

native is the modifier applicable only for methods and we cannot apply anywhere else.

The methods which are implemented in non java (mostly C or C++) are called native methods or foreign methods

The main objective native keywords 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

Pseudo code to use native keyword in java

```
Class Native{
    Static{
        System.loadLibrary("native library path"); → 1. Load native libraries
    }
    Public native void m1 ();
} → 2. Declare a native method

Class Client{
    Public static void main(String[ ] args){
        Native n = new Native();
        n.m1(); → 3. Invoke a native method
    }
}
```

For native methods implementations is already available in old languages like C or C++ and we are responsible to provide implementation hence native method declaration should end with semicolon(;) .

Public native void m1(); → valid

Public native void m1 (){ } → invalid CE: native methods cannot have a body

For native methods implementation is already available in old languages but for abstract methods implementation should not be available hence we cannot declare native method as abstract that is native abstract combination is illegal combination for methods.

We cannot declare native method as strictfp () because there is no guarantee that old languages follow IEEE754 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.

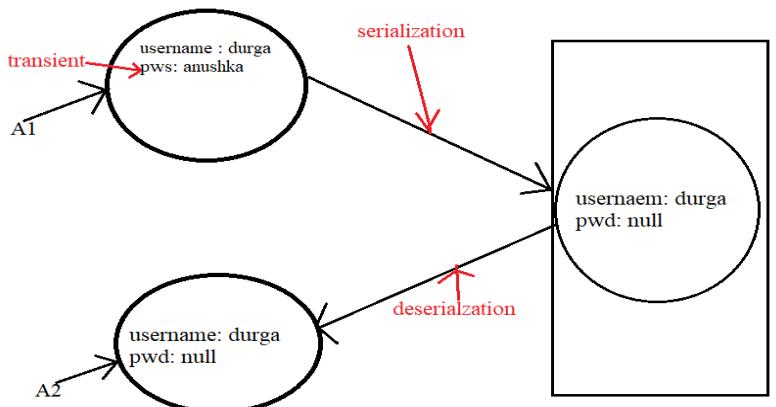
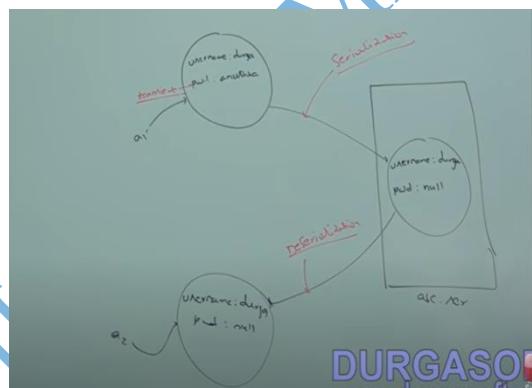
Transient keyword

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 the variable as transient.

At the time of serialization JVM ignores original value of transient variable and save default value to the file hence **transient means not to serialize** .



Volatile modifier

Volatile is the modifier applicable only for variable and we cannot apply anywhere else

If the value of a variable keep on changing by multiple threads then there may be 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

Every modification performed by the thread will takes place in local copy so that there is no effect on 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 maintain 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 always deprecated keyword.

Final variable means the value never changes whereas volatile variable means the value keep on changing hence volatile final is illegal combination for variables.

modifier	classes		methods	variables	blocks	interfaces		Enum		constructors
	outer	inner				outer	inner	outer	inner	
public	yes	yes	yes	yes	no	yes	yes	yes	yes	yes
private	no	YES	yes	yes	no	no	YES	no	YES	yes
protected	no	YES	yes	yes	no	no	YES	no	YES	yes
<default>	yes	yes	yes	yes	no	yes	yes	yes	yes	yes
final	yes	yes	yes	yes	no	no	no	no	no	no
abstract	yes	yes	yes	no	no	yes	yes	no	no	no
static	no	YES	yes	yes	yes	no	YES	no	YES	no
synchronized	no	no	yes	no	yes	no	no	no	no	no
native	no	no	yes	no	no	no	no	no	no	no
strictfp	yes	yes	yes	no	no	yes	yes	yes	yes	no
transient	no	no	no	yes	no	no	no	no	no	no
volatile	no	no	no	yes	no	no	no	no	no	no

- ✓ The only applicable modifier for local variable is final.
- ✓ The only applicable modifiers for constructor public, private, Protected and default
- ✓ The modifier which is applicable for only methods is native
- ✓ The modifier which are applicable only for variables volatile and transient
- ✓ The modifier which are applicable for classes but not for interface is final
- ✓ The modifier which are applicable for classes but not for enum final and abstract

Interfaces

1. Introduction
2. Interface declaration & implementation
3. Extends V/S implements
4. Interfaces methods
5. Interface variables
6. Interface naming conflicts
 - 6.1. Method naming conflicts
 - 6.2. Variable naming conflicts
7. *** Marker interface
8. Adapter classes
9. Interfaces V/S abstract class V/S concrete class
10. *** Difference between interfaces and abstract class
11. Conclusion

Definition 1:

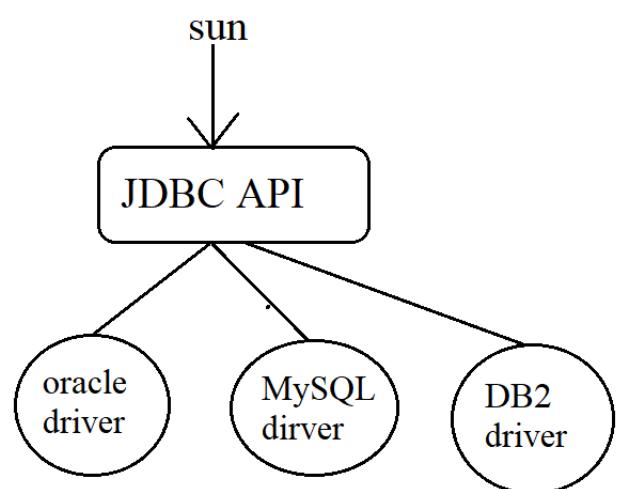
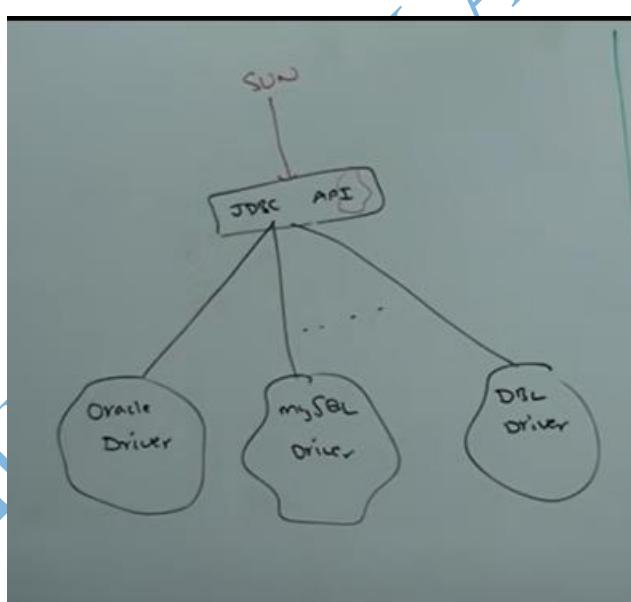
Any service requirement specification (SRS) is considered as an interface.

Example 1:

JDBC API acts as requirement specification to develop data base driver

Database vendor is responsible to implement this JDBC API

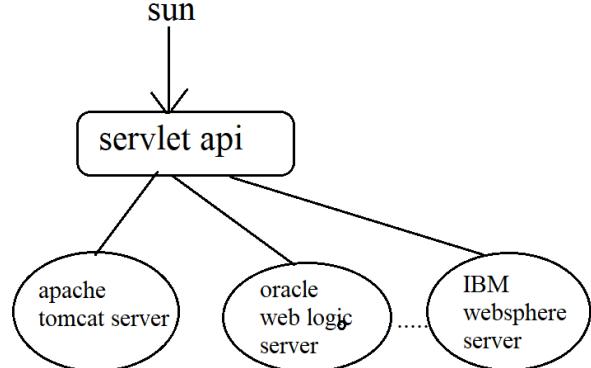
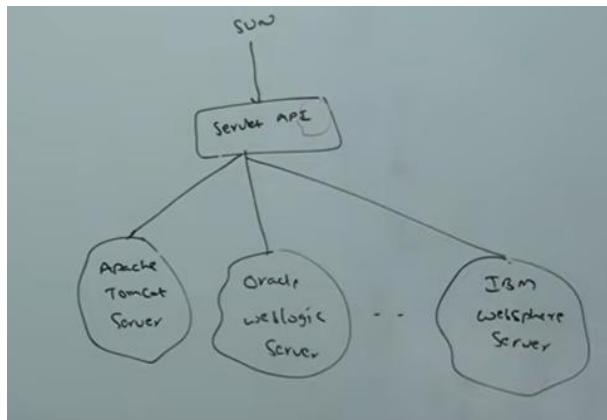
+



Example 2:

Servlet api acts as requirement specification to develop web server

Web server vendor is responsible to implement servlet api



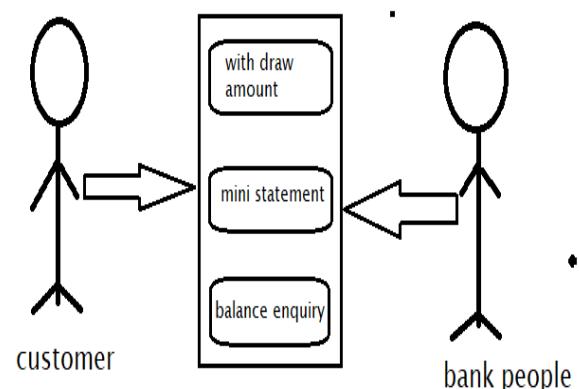
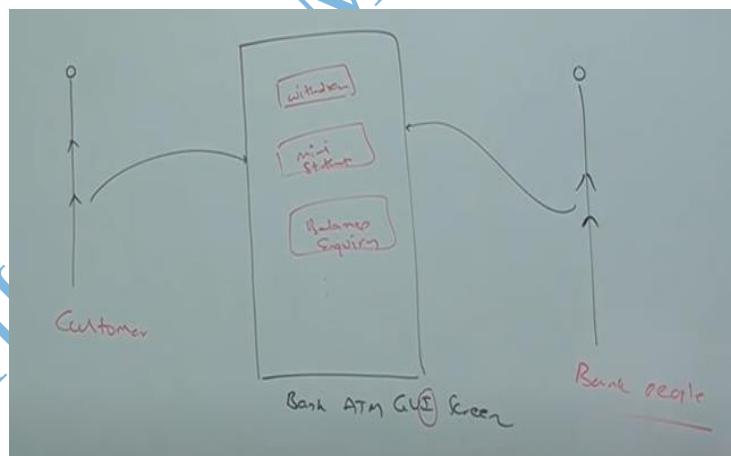
Definition 2:

From client point of view an interface defines the set of services what he is expecting

From service provider point of view an interface the set of services what he is offering hence **any contract between client and service provider is considered as an interface.**

Example:

Through bank ATM GUI screen bank people are highlighting the set of service of they are offering at the same time the GUI screen represents the set of services what customer expecting hence this GUI screen acts as contract between customer and bank people



Definition 3:

Inside interface every method is always abstract whether we are declaring or not hence interface is considered 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.

Interface declaration & implementation

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 abstract then next level child class is responsible to provide implementation

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 compile time error

Example:

```
interface Interf
{
    Void m1();
    Void m2();
}

abstract class ServiceProvider implements Interf
{
    public void m1()
    {
    }
}

class SubServiceProvider extends ServiceProvider
{
    public void m2 ()
    {
    }
}
```

Extends V/S implements

1. A class can extend only one class at a time
2. An interface can extend any number of interfaces simultaneously

Example:

```
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 interface simultaneously

Example:

```
Class A extends B implements C,D,E
{
}
```

Which of the following is valid?

- | | |
|---|-----------|
| 1. A class can extend any number classes at a time | → invalid |
| 2. A class can implement only one interface at a time | → invalid |
| 3. A interface can extend only one interface at a time | → invalid |
| 4. A interface can implement any number interfaces simultaneously | → invalid |
| 5. A class can extend another class or can implement an interface but not both simultaneously | → invalid |
| 6. Non of the above | → valid |

Consider the following expression **X extends Y** for which of the following possibility's of X and Y 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 interface →valid**
4. No restriction

X extends Y,Z

→ X, Y,Z should be interface

X implements Y,Z

→ X → class

→ Y,Z → interfaces

X extends Y implements Z

→ X, Y → classes

→ Z → interface

X implements Y extends Z

→ CE: because we have to take extends first follower by interface

Interface methods

1. Every method present inside interface is always public and abstract whether we are declaring are not.

Interface Interf

```
{  
    ..... Void m1 ();  
}
```

.... **Public:** to make this method available to every implementation class

..... **Abstract:** implementation class is responsible to provide implementation

2. Hence inside interface the following method declarations are equal

```
Void m1();  
Public void m1();  
Abstract void m1();  
Public abstract void m1();
```

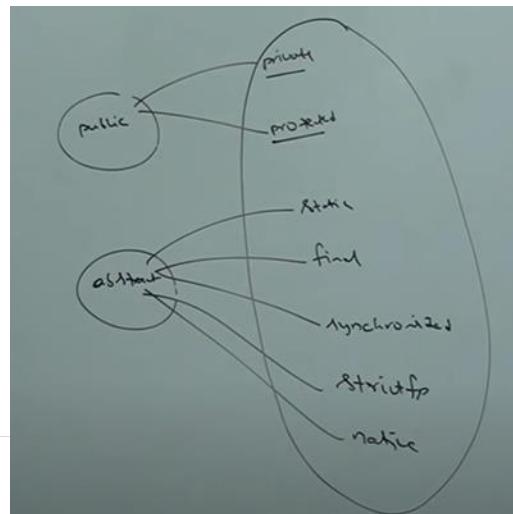
3. As every interface method is always public and abstract we cannot declare interface method with the following modifiers.

Public

Private
Protected

Abstract

Static
Final
Synchronized
Strictfp
Native



Which of the following method declarations are allowed inside interface?

- | | |
|--------------------------------------|-----------|
| 1. Public void m1(){} | →invalid |
| 2. Private void m1(); | → invalid |
| 3. Protected void m1(); | → invalid |
| 4. Static void m1(); | → invalid |
| 5. Public abstract native void m1(); | → invalid |
| 6. Abstract public void m1(); | →valid |

Interface variables

1. An interface can contain variables the main purpose of interface variable is to define requirement level constants.
2. Every interface variable is always public static final whether we are declaring are 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 affected. To restrict this every interface variable is always final

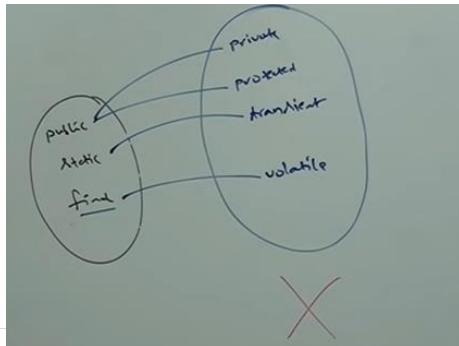
3. Hence within the interface the following variable declarations are equal

```
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;
```

equal

4. As every interface variable is always public static final we cannot declare with the following modifiers

Public	
Private	
Protected	
Static	
Transient	
Final	
Volatile	



5. For interface variable compulsory we should perform initialization at the time of declaration otherwise we will get compile time error

```
Interface Interf
{
    Int x;
}
CE: = expected
```

Inside interface which of the following variable declaration are allowed?

- 1. Int x; → invalid
- 2. Private int x = 10; → invalid
- 3. Protected int x =10; → invalid
- 4. Volatile int x = 10; → invalid
- 5. Transient int x =10; → invalid
- 6. Public static int x =10; → valid

Inside implementation class we can access interface variables but we cannot modify values

```
Interface Interf
{
    Int x = 10;
}
```

```
Class Test implements Interf
{
    Public static void main(String[ ] args)
    {
        X = 777;      → CE: cannot assign a value to final variable X
        System.out.println(x)
    }
}
Class Test implements Interf
{
    Public static void main(String[ ] args)
    {
        Int x = 777;
        System.out.println(x); → o/p: 777
    }
}
```

Interface naming conflicts

Method naming conflicts:

Case 1:

If two interface contains a method with same signature and same return type then in the implementation class we have to provide implementation for only one method

Example:

```
Interface Left
{
    Public void m1();
}
```

```
Interface Right
{
    Public void m1();
}
```

```
Class Test implements Left,Right
{
    Public void m1()
    {
        }
}
```

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 method acts as overloaded methods.

Example:

```
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 contain a method with same signature but different return types then it is impossible to implement both interfaces simultaneously (if return types are not covariant)

Example:

Interface Left

```

{
    Public void m1 ( );
}

```

Interface Right

```

{
    Public void m1();
}

```

We can't write any java class which implements both interface simultaneously

Is a java can implement any number of interfaces simultaneously?

Yes, except a particular case if two interface contains a method with same signature but different return types then it is impossible to implement both interfaces simultaneously

Interface 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

Example:

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);      → CE : reference to x is ambiguous
        System.out.println(Left.x);   → 777
        System.out.println(Right.x);  →888
    }
}

```

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

Example:

- 1. Serializable(I),
 - 2. Cloneable(I),
 - 3. RandomAccess (I),
 - 4. SingleThreadMode(I);
- These are marked for some ability

Example 1:

By implementing serializable (I) our objects can be saved to the file and can travel across the network.

Example 2:

By implementing cloneable(I) our objects are in a position to produce duplicate cloned objects.

Without having any methods how the objects will get some ability in marker interfaces?

Internally JVM is responsible to provide required ability

Why JVM is providing required ability in marker interfaces?

To reduce complexity of programming and to make java language as simple

Is it possible to create our own marker interface?

Yes, but customization of JVM is required.

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();
    ....
    ....
    ....
    M1000();
}
```

If we implement an interface for each and every method of that interface compulsory we should provide implementation whether it is required or not required

```
Class Test implements X
{
    M3()
    {
        10 lines of code;
    }
    M1();
    M2();
    M4();
}
```

The problem in this approach is it increases the length of the code and reduces readability.
We can solve this problem by using adapter class.

Instead of implementing interface we can extend adapter class we have to provide implementation only for required methods and we are not responsible to provide implementation for each and every method of the interface so that length of the code will be reduced.

```

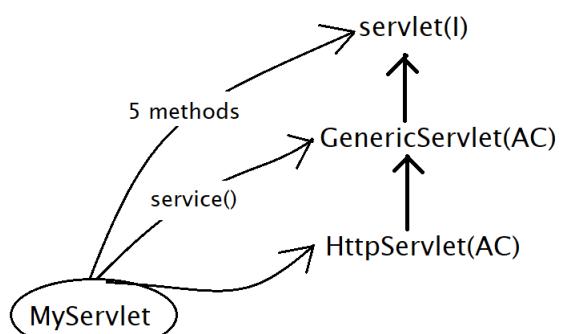
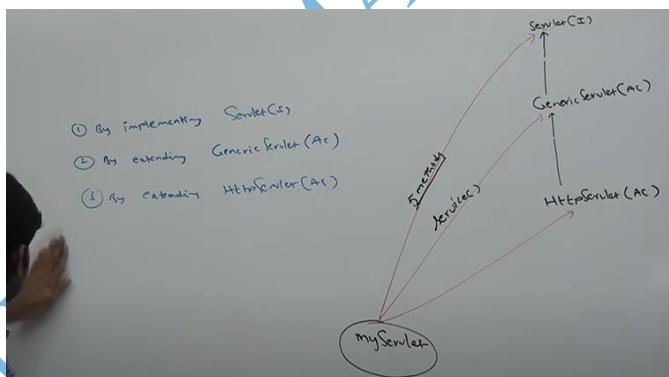
Class Test extends adapter
{
    M3()
    {
        .....
        .....
    }
}
Class Sample extends adapter
{
    M7()
    {
        .....
        .....
    }
}
Class Demo extends adapter
{
    M1000()
    {
        .....
        .....
    }
}

```



We can develop a servlet in three ways

1. By implementing servlet(I)
2. By extending Generic Servlet(AC)
3. By extending HttpServlet(AC)



If we implement servlet interface for each and every method of that interface we should provide implementation it increases length of the code and reduces readability.

Instead of implementing servlet interface directly if we extend generic servlet we have to provide implementation only for service method and for all remaining method we are not required to provide implementation hence more or less generic servlet acts as adapter class for servlet interface.

Note:

Marker interface and adapter classes simplify complexity of programming and these are best utilities to the programmer and programmer life become simple.

Interface V/S abstract class V/S concrete class

If we don't know anything about implementation just we have requirement specification then we should go for interface.

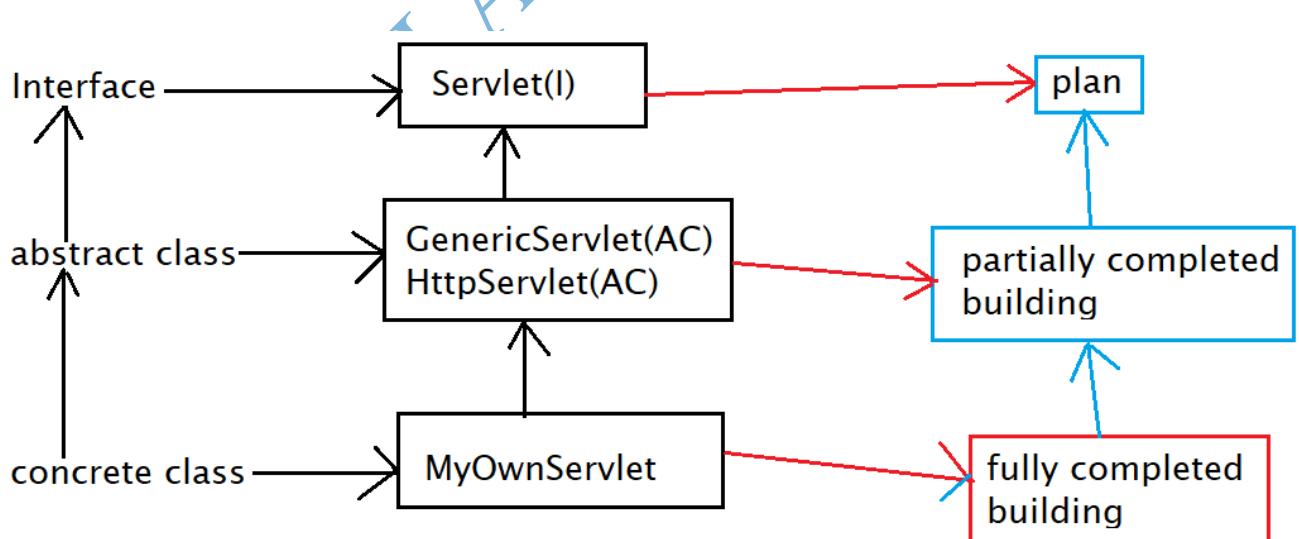
Example: servlet

If we are taking about implementation but not completely (partially implementation) then we should go for abstract class.

Ex: GenericServlet , HttpServlet

If we are taking about implementation completely and ready to provide service then we should go for concrete class.

Ex: MyOwnServlet



Difference between interface and abstract

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 in abstract class need not public and abstract and we can take concrete methods also
3. as every interface method is always public and abstract and hence we cannot declare with the following modifiers Private, protected, final, static, synchronized, native and strictfp..	3. there are no restriction on abstract class method modifiers
4. every variable present interface is always public, static, final whether we are declaring or not	4. every variable present in abstract class need not be public, static, final
5. as every interface variable is always public ,static, final we cannot declare with the following modifiers private, protected, volatile and transient.	5. there are no restriction on abstract class variable modifiers
6. for interface variables compulsory we should perform initialization at the time of declaration only other wise compile time error	6. for abstract class variables we are not required to perform initialization at the time of declaration.
7. inside interface we can't declare static and instance blocks	7. inside abstract class we can declare static and instance blocks
8. inside interface we can't declare constructors	8. inside abstract class we can declare constructors.

Any way 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

```
Abstract class Person{  
    String name;  
    Int age;  
    ....  
    ....  
    ..... (100 properties)  
}
```

```

Class Student extends Person {
    Int rollno;
    Student(String name, int age, .....)// 101 properties
    {
        This.name = name;
        This.age = age;
        .....
        .....
        (100 properties)
        .....
        This. Rollno = rollno;
    }
}

```

Student s1 = new Student(101 properties);

```

Class Teacher extends Person
{
    String subject;
    Teacher(String name, int age, .....)// 101 properties
    {
        This.name = name;
        This.age = age;
        .....
        .....
        (100 properties)
        .....
        This. Subject = subject;
    }
}

```

Teacher t = new Teacher(101 properties)

(more code, code redundancy)

Approach 2: With constructor inside abstract class

```

Abstract class Person
{
    String name;
    Int age;
    .....
    .....
    .... (100 properties)
Person (String name, int age, .....){
    This.name = name;
    This.age = age;
    .....
    .....
    .... (100 lines of code)
} (this constructor will work for every child object initialization)
}

```

```

Class Student extends Person
{
    Int rollno;
    Student(String name, int age, .....)// 101 properties
    {
        Super(100 properties);
        This. Rollno = rollno;
    }
}

```

Student s1 = new Student(101 properties);

```

Class Teacher extends Person
{
    String subject;
    Teacher(String name, int age, .....)// 101 properties
    {
        Super(100 properties)
        This.Subject = subject;
    }
}

```

Teacher t = new Teacher (101 properties)

(1000 child classes)

(Less code and reusability)

Note:

Either directly or indirectly we can't create an object for abstract class.

Any way we cannot create objects for abstract class and interface but abstract class can contain constructor but interface doesn't contain constructor what is the reason?

- ✓ The main purpose of constructor is to perform initialization for the instance variables.
- ✓ Abstract class can contains instance variables which are required for child object.
- ✓ To perform initialization of those instance variable 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
- ✓ Whenever we are creating child class object parent object won't be created just parent class constructor will be executed for the child object purpose only

```

Class Parent
{
    Parent()
    {
        System.out.println(this.hashCode());
    }
}
Class Child extend Parent
{
    Child()
    {
        System.out.println(this.hashCode());
    }
}
Class Test
{
    Public static void main(String[ ] args)
    {
        Child c = new Child();
        System.out.println(c.hashCode());
    }
}

```

Inside interface every method is always abstract and we can take only abstract methods in abstract also then what is the difference between interface and abstract class that is 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 sweeping activity
- ✓ If everything is abstract then it 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 and hence we are missing inheritance benefit

In this case object creation is costly

Example :

Test T = new Test(); 2min

Approach -2

Interface x

{

}

Class test implements X

{

}

While implementing interface we can extend some other class hence we won't miss any inheritance benefit

In this object creation is not costly

Example

Test T = new Test(); 2 seconds

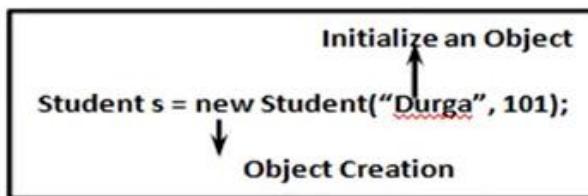
New v/s constructor:

Roles of new keyword and constructor

1. Whenever we are creating child class object whether parent object will be created or not?
2. Whenever we are creating child class object, what is the need of executing parent class constructor?
3. Any way we cannot create object for abstract class either directly or indirectly, but abstract class can contain constructor. What is need?
4. Any way we cannot create object for abstract class and interface. Abstract class can contain constructor but interface does not. Why?
5. Inside interface we can take only abstract methods. But in abstract class also we can take only abstract methods based on our requirement. Then what is the need of interface? That is is it possible to replace interface concept with abstract class?

The purpose of constructor is

1. To initialize an object but not to create an object
2. Whenever we are creating an object after object creation automatically constructor will be executed to initialize that object.
3. Object creation by new operator and then initialization by constructor.



4. Before constructor only object is ready and hence within the constructor we can access object properties like Hash code.

Example:

```
Class Test
{
    Test( )
    {
        System.out.println(this); // Test@623d60
        System.out.println(this.hashCode()); // 7224672
    }
    Public static void main(String [ ] args)
    {
        Test t = new Test();
    }
}
```

The main objective of new operator is to create an object

The main purpose of the constructor is to initialize the object

First object will be created by using new operator and then initialization will be performed by constructor

Example:

Class student

```
{  
    String name;  
    Int rollno;  
    Student (String name, int roolno)  
    {  
        This.name = name;  
        This.rollno = rollno;  
    }(constructor)  
}
```

Student s = new Student("durga",101)

responsible to create object

to initialize object

1. Child object V/s parent Constructor

Whenever we are creating child class object automatically parent constructor will be executed but parent object won't be created

The purpose of parent constructor execution is to initialize child object only. Of course for the instance variables which are inheriting from parent class.

Class Parent

```
{  
    Parent()  
    {  
        system.out.println(this.hashCode()); //7224672  
    }  
}
```

Class Child extend Parent

```
{  
    Child()  
    {  
        System.out.println(this.hashCode()); //7224672  
    }  
}
```

Class Test{

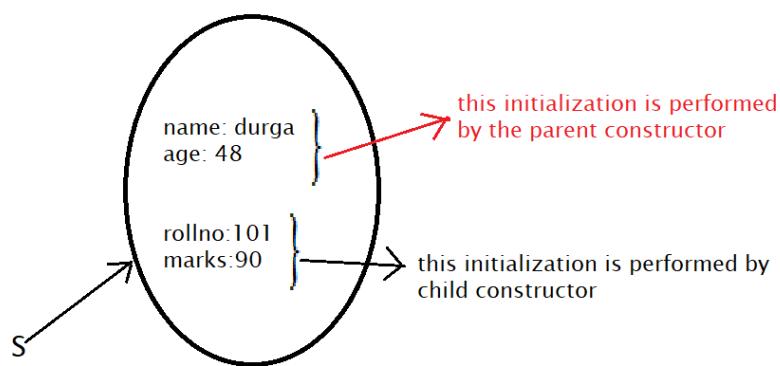
```
    Public static void main(String[ ] args){  
        Chid c = new Child();  
        System.out.println(c.hashCode()); //7224672  
    }  
}
```

In the above example whenever we are creating child class object but both parent and child constructors will be executed for that child class object.

Answer 2:

Whenever we are creating child class object automatically parent constructor will be executed to perform initialization for the instance variables which are inheriting from parent

```
Class Person
{
    String name;
    Int age;
    Person(String name, int age)
    {
        This.name = name;
        This .age = age;
    }
}
Class Student extends Person
{
    Int rollno;
    Int marks;
    Student(String name int age, int rollno,int marks)
    {
        Super(name,age);
        This .rollno = rollno;
        This .marks= marsk;
    }
}
Student s = new Student ("durga", 48,101,90);
```



In the above program both parent and child constructors executed for child object initialization only

3 answer:

Anyway we can not create object for abstract class either directly or indirectly, but abstract class can contain constructor what is the need.

The main objective abstract class constructor is to perform initialization for the instance variable which are inheriting from abstract class to the child class.

Whenever we are creating child class object automatically abstract class constructor will be executed to perform initialization for the instance variable which is inheriting from abstract class(code reusability).

Without constructor in abstract class:

Abstract Class Person

```
{  
    String name;  
    Int age;  
  
    (100 properties)  
}
```

Class Student extend Person

```
{  
    Int rollno;  
    Int marks;  
    Student(String name, int age, int rollno, int marks)  
    {  
        This.name = name;  
        This.age= age;  
        This.rollno = rollno;  
        This.marks = marks;  
    }  
}  
Student s1 = new Student("durga",48,101,90);
```

Class Teacher extend Person

```
{  
    Double salary;  
    String subject;  
    Teacher(String name ,int age, dounble salary, String subject)  
    {  
        This.name = name;  
        This.age = age;  
        Double salary = salary;  
        String subject = subject;  
    }  
}
```

```
Teacher t = new Teacher("nagoor",45,25,java)
```

With constructor in abstract class

```
Abstract Class Person
```

```
{  
    String name;  
    Int age;  
    Person(String name, int age)  
    {  
        This.name = name;  
        This.age = age;  
    }  
}
```

```
Class Student extend Person
```

```
{  
    Int rollno;  
    Int marks;  
    Student(String name, int age, int rollno, int marks)  
    {  
        Super(name,age);  
        This.rollno = rollno;  
        This.marks = marks;  
    }  
}  
Student s1 = new Student("durga",48,101,90);
```

```
Class Teacher extend Person
```

```
{  
    Double salary;  
    String subject;  
    Teacher(String name ,int age, dounble salary, String subject)  
    {  
        Super(name,age);  
        Double salary = salary;  
        String subject = subject;  
    }  
}  
Teacher t = new Teacher("nagoor",45,25,java)
```

4. abstract class v/s interface with respect to constructor

Anywhere we cannot create object for abstract and interface but abstract class can contain constructor but interface doesn't why?

The main purpose of constructor is to perform initialization of an object that is to perform initialization for instance variables

Abstract class can contain instance variable which are required for child class object to perform initialization for these instance variables constructor concept is required for abstract classes

Ever variable present inside interface is always public static final whether we are declaring or not hence there is no chance of existing instance of variables inside interface because of this constructor concept not required for interfaces.

```
Abstract Class Person
{
    String name;
    Int age;
    Person(String name, int age)
    {
        This.name = name;
        This.age = age;      → current child class object
    }
}
```

Inside interface we can take only abstract methods. But in abstract class also we can take only abstract methods based on our requirement. Then what is the need of interface? That is is it possible to replace interface concept with abstract class?

If everything is abstract is highly recommended to go for interface but not abstract class

We can replace interface with abstract class but it is not good programming practice (this is something like recruiting IAS officer for sweeping purpose)

Interface X	Abstract class X
{ }	{ }
While implementing interface we can extend any other class and hence we won't miss inheritance benefit	While extending abstract class we cannot extend any other class and hence we are missing inheritance benefit.
Class Test extends A implements X { (valid) }	Class Test extends X,A { (invalid) }

<p>In this case object creation is not costly Class Test implements X</p> <pre>{ } Test t = new Test(); 2 min</pre>	<p>In this case object creation is costly Class Test extends X</p> <pre>{ } Test t= new Test(); 20 min</pre>
---	--

Summary:

Which of the following are valid?

1. The purpose of constructor is to create an object.
→ invalid
2. The purpose of constructor is to initialize an object but not to create object.
→ valid
3. Once constructor completes then only object creation completes.
→invalid
4. First object will be created and then constructor will be executed.
→ valid
5. The purpose of new keyword is to create object and the purpose of constructor is to initialize that object
→ valid
6. We can't create object for abstract class directly but indirectly we can create. →invalid
7. Whenever we are creating child class object automatically parent class object will be created internally.
→invalid
8. Whenever we are creating child class object automatically abstract class constructor will be execute.
→ valid
9. Whenever we are creating child class object automatically parent object will be created.
→invalid
10. Whenever we are creating child class object automatically parent constructor will be executed but parent object won't be created.
→valid
11. Either directly or indirectly we can't create object for abstract class or hence constructor concept is not applicable for abstract class.
→ invalid
12. Interface can contain constructor. →invalid

Oops(object oriented programings)

Moudule -1

1. Data hiding
2. Abstraction
3. Encapsulation
4. Tightly encapsulated class

Module -2

5. IS – A relationship
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

Data hiding

Outside person cannot access our internal data directly or our internal should not go out directly this OOP feature is nothing but data hiding.

After validation or authentication outside person can access our internal data

Example:

After providing proper username and password we can able to access our gmail inbox information

Example 2:

Even though we valid customer of the bank we can able to access our account information and cannot access others account information.

By declaring data member (variable) as private we can achieve data hiding

Example:

```
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 member (variable) as private

Abstraction

Hiding internal implementation and just highlight the set of service what we are offering is the concept of abstraction.

Through the bank ATM GUI screen bank people or 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 effecting outside person we can able to perform any type of change in our internal system and hence enhancement will become easy.
3. It improves maintainability of the application
4. It improves easiness to use our system.

By using interface and abstract classes we can implement abstraction

Encapsulation

The process of binding data and corresponding methods into a single unit is nothing but encapsulation.

Example:

Class Student

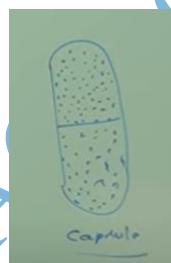
{

 Data members;

 +

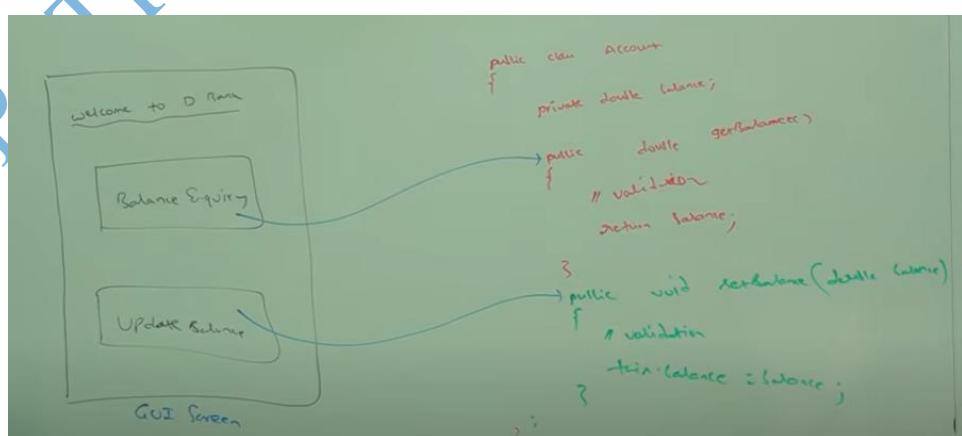
 Methods(behaviour); sni

}



If any component follows data hiding and abstraction such type of component is said to be encapsulated component.

Encapsulation = data hiding + abstraction



Example:

```
Public class Account
{
    Private double balance;
    Public doubl getBalance()
    {
        // validation
        Return balance;
    }
    Public coid setBalance(double balance)
    {
        //validatioin
        This.balance= balance;
    }
}
```

The main advantages of encapsulations are:

1. We can achieve security
2. Enhancement become easy
3. It improves maintainability of the application

The main advantage of encapsulation is we can achieve security but the main disadvantage of encapsulation is it increases length of the code and slows down execution.

Tightly encapsulated class

A class is set to be tightly encapsulated if and only if each and every variable declared as private.

Whether class contains corresponding getter and setter methods are not and whether these methods are declared as public are not these things we are not required to check.

Example:

```
Public class Account
{
    Private double balance;
    Public double getBalance()
    {
        Return balance;
    }
}
```

Which of the following classes tightly encapsulated?

Class A
{
 Private int x = 10;
}(valid)

Class B extends A
{
 Int y =20;
}(invalid)

Class C extends A
{
 Private int z = 30;
}(valid)

Which of the following classes are tightly encapsulated?

Class A

```
{  
    Int x = 10;  
}(invalid)
```

Class B extends A

```
{  
    Private int y = 20;  
}(invalid)
```

Class C extends B

```
{  
    Private int z = 30;  
}(invalid)
```

Note: if the parent class non tightly encapsulated then no child class is tightly encapsulated

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 key word we can implement IS-A relationship

Class Parent

```
{  
    Public void m1()  
    {  
        System.out.println("parent");  
    }  
}
```

Class Child extends Parent

```
{  
    Public void m2()  
    {  
        System.out.println("child");  
    }  
}
```

Class Test

```
{  
    Public static void main(String[ ] args)  
    {  
        1. Parent p = new Parent();  
            p.m1();      → valid  
            p.m2();      → invalid CE: connot find symbol symbol: method m2(),  
location : class Parent
```

```

2. Child c = new Child();
   c.m1();      → valid
   c.m2();      → valid

** 3. Parent p1= new Child();
       p1.m1();    → valid
       p1.m2();    → invalid CE:cannot find symbol; symbol : method
m2(), location class p;

4. Child c1 = new Parent(); → 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 cannot call child specific methods
3. Parent reference can be used to hold but using that reference we cannot call child specific methods but we can call the methods present parent class.
4. Parent reference can be used to hold child object but child reference cannot be used to hold parent object.

Without inheritance

```

Class VLoan
{
  300 methods
}
Class HLoan
{
  300 methods
}
Class PLoan
{
  300 methods
}
900 methods
90 hours

```

With inheritance

```

Class Loan
{
  250 methods
}

Class VLoan extends Loan
{
  50 methods
}

Class HLoan extends Loan
{
  50 methods
}

Class PLoan extends Loan
{
  50 methods
}

400 methods 40 hours

```

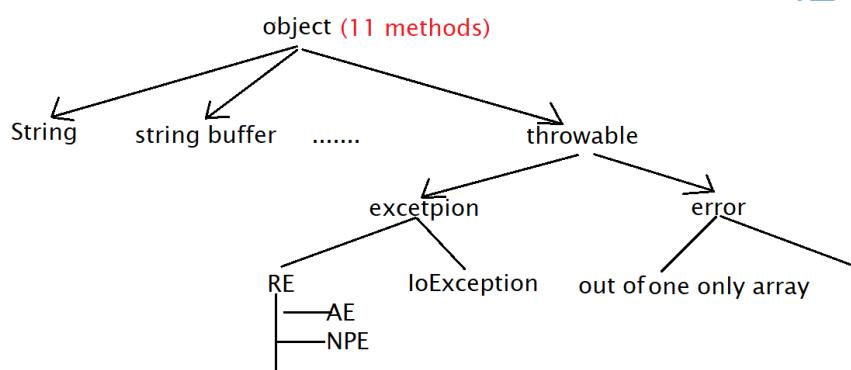
Note:

The most common methods which are applicable for any type of child, we have to define in parent class the specific method which are applicable for a particular child we have to define in child class.

Total java API is implemented base on inheritance concept

The most common methods which are applicable for any java object or defined in object class and hence every class in java is the child class of object either directly or indirectly hence so that object class method by default available for every java class without rewriting due to this object class act as root for all java classes

Throwable class defines the most common methods which are required for every exception and error classes hence these class acts as root for java exception hierarchy



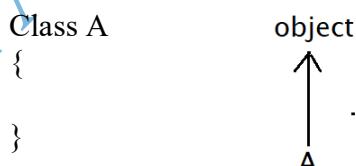
Multiple inheritances:

A java class cannot extend more class at a time hence java won't provide for multiple inheritance

Class A extends B, C
{
 (invalid)
}

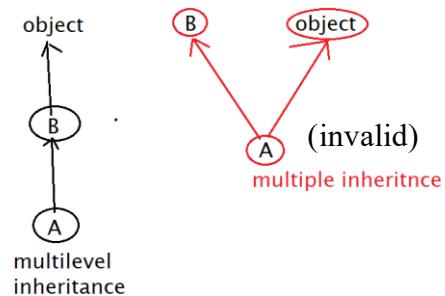
Note:

1. If our class doesn't extend any other class then only our class is direct child class of object



2. If our class extends any other class then our class is indirect child class of object

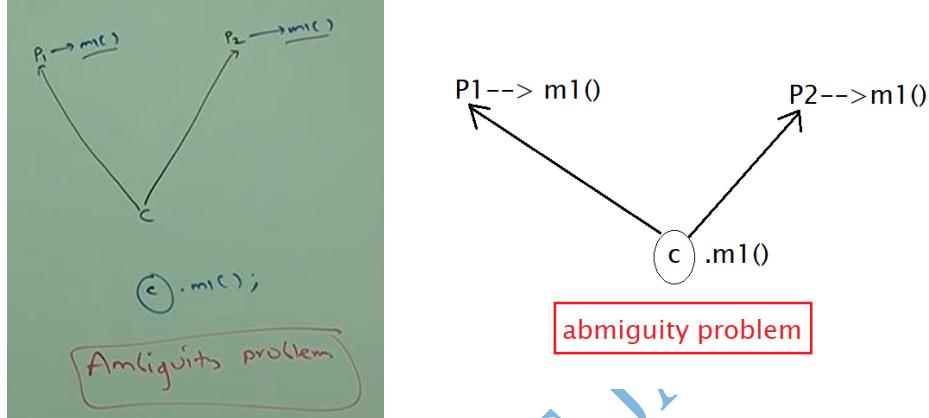
Class A extends B
{
}



3. Either directly or indirectly java won't provide for inheritance with respect to classes

Why java won't provide support for multiple inheritances?

There may be a chance of ambiguity problem hence java provide support multiple inheritance

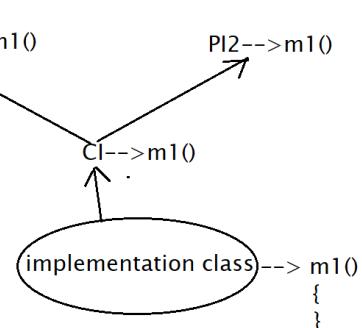


On the child object if I call `m1()` may be ambiguity problem

but interface can extend any number of interfaces simultaneously hence java provide supports for multiple inheritance with respect to interfaces

```
interface A
{
}
Interface B
{
}
Interface C extends A,B
{
}
```

Why ambiguity problem won't be there in interfaces?



Even though multiple method declaration 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

Cyclic inheritance:

Cyclic inheritance is not allowed in java off course it is no required

Example

```
Class A extends A  
{  
}  
}
```

```
Class A extends B  
{  
}  
}  
Class B extends A  
{  
}  
}
```

CE: cycle inheritance involving A

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 the most of the times we depending on new keyword.
3. The main advantage of has a relationship is reusability of the code

Example:

```
Class Engine  
{  
    //engine specific functionality;  
}  
Class Car  
{  
    Engine e = new engine();  
}
```

Car Has -A engine reference

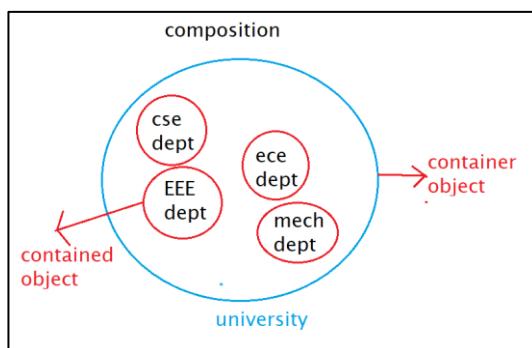
Difference between composition and aggregation?

Composition:

Without existing container object if there is no chance of existing contained objects then container and contained objects strongly associated and this strong association is nothing but composition.

Example:

University consists of several departments without existing university there is no chance of existing department hence university and department strongly associated and this strong association is nothing but composition

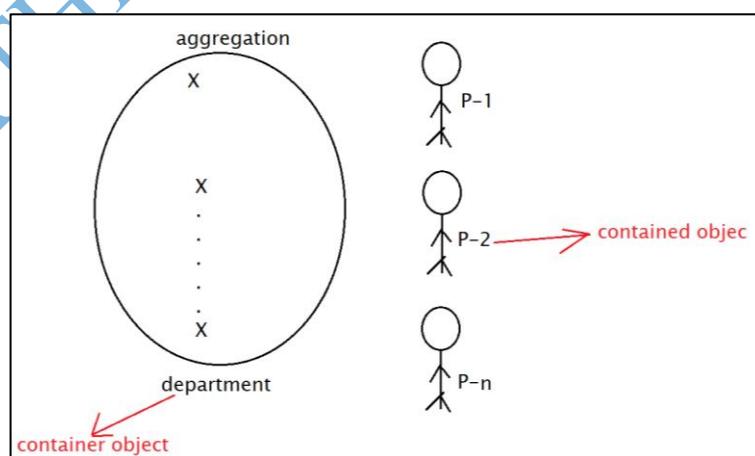


Aggregation:

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.

Example:

Department consists several professors without existing department there may be a chance of existing professor object hence department and professor objects are weakly associated and this week association is nothing but aggregation.



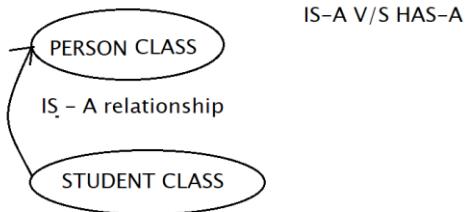
Note:

1. In composition objects are strongly associated where as in aggregation objects are weakly associated.
2. In composition container object holds directly contained objects where as in aggregation container object holds just reference of contained objects.

IS-A V/S HAS-A:

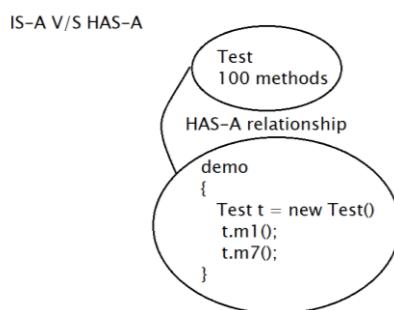
If we want total functionality of a class automatically then we should go for IS-A relationship.

Example:



If we want part of the functionality then we should go for HAS-A relationship

Example:



Method signature:

In java method signature consists of method names followed by argument types

Example:

public static int m1(int i, float f) → m1(int, float)

return type is not a part of method signature in java.

Compiler will use method signature to resolve method calls
Class test

M1(int)
M2(String)

Method Table

```

Example:
Calss Test
{
    Public void m1(int i)
    {

    }
    Public void m2(String s)

    }
}

```

```

Test t = new Test();
t.m1(10);           →valid
t.m2("malachi");   →valid
t.me(10.5)          → CE: cannot find symbol; symbol: method m3(double); location:
                           class Test

```

within a class two method with same signature is not allowed

```

example:
class Test
{
    Public void m1(int i) → m1(int)
    {
    }
    Public void m1(int x) →m1(int)
    {
        Return 10;
    }
}
Test t = new Test();
t.m1(10);

```

CE: m1(int) is already defined in Test

Overloading

Two methods are said to be overloaded if and only **if both methods having same name with different arguments 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 compulsory we should go for new method name which increases complexity of programming.

Example: 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.

Example:

```
abs(int i);  
abs(long g);  
abs(float f);
```

overloaded methods

Having overloading concept in java reduces complexity of programming.

Example:

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");  
    }  
    Public static void main(String[ ] args)  
    {  
        Test t = new Test();  
        t.m1();          //no-arg  
        t.m1(10);       //int-arg  
        t.m1(10.5);    //double-arg  
    }  
}
```

*** in overloading method resolution always takes care by compiler based on reference type hence overloading is also consider as **compile time polymorphism or static polymorphism or early binding**

Case – 1:

Automatic promotion in overloading

While resolving overloaded methods if exact method is not available then we won't get any compile time error immediately 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 the argument once again to the next level this process will be continued until all possible promotions still if the matched method is not available then we will get compile time error the following are all possible promotion in overloading

Byte → short → int → long → float → double
Char → int

This process is called automatic promotion in overloading

Example:

```
Class test
{
    Public Void m1(int i)
    {
        System.out.println("int-arg");
    }
    publicVoid m1(float f)
    {
        System.out.println("int-arg");
    }
    Public static void 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(10l);         //float-arg
        t.m1(10.5);
        //CE: cannot find symbol; symbol: method m1(double); location class Test
    }
}
```

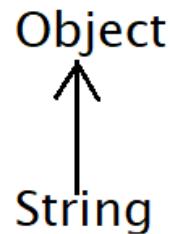
Case – 2:

Example:

```
Class Test
{
    Public vod m1(String s)
    {
        System.out.println("String
version");
    }
    Public void m1(object o)
    {
        System.out.println("Object
version");
    }
}
```

```
}}

Public static void main(String[ ] args)
{
    Test t = new Test();
    t.m1(new Object());
    // object version
    t.m1("durga");
    //String version
    t.m1(null);
    //string version
}
}
```



While resolving overloaded methods compiler will always give the precedence for child type argument when compared with parent type argument

Case – 3:

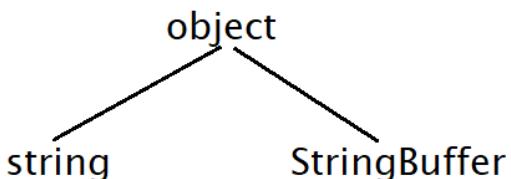
Example:

```
Class Test
{
    Public vod m1(String s)
    {
        System.out.println("String
version");
    }
    Public void m1(StringBuffer sb)
    {
        System.out.println(" StringBuffer
version");
    }
}
```

```
}
```

```
Public static void main(String[ ] args)
{
    Test t = new Test();

    t.m1("durga");
    //String version
    t.m1(new StringBuffer("durga"));
    //StringBuffer version
    t.m1(null);
    //CE: reference to m1() is ambiguous
}
```



Case – 4:

Example:

```
Class Test
{
    Public vod m1(int I, float f)
    {
        System.out.println("int - float
version");
    }
    Public void m1(float f, int i)
    {
        System.out.println("StringBuffer
version");
    }
}

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

```
Test t = new Test();

t.m1(10,10.0f);
//int-float version
t.m1(10.5f,10);
//float-in version
t.m1(10,10);
//CE:reference to m1() is
ambiguous
t.m1(10.0f, 10.5f);
//CE: cannot find symbol; symbol:
method m1(float, float); location: class
Test
}
```

Case – 5:

Example:

```
Class Test
{
    Public vod m1(int x)
    {
        System.out.println("general
method");
    }
    Public void m1(int...x)
    {
        System.out.println(" var-arg
method");
    }
}
```

```
Public static void main(String[ ] args)
{
    Test t = new Test();

    t.m1();      //var-arg method
    t.m1(10,20); // var-arg
method

    t.m1(10);   //general method
}
```

In general var-arg method will get least priority that is no other method matched then only var-arg method will get the chance it is exactly same as default case inside switch.

Case – 5:

```
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");
    }
}
```

```
Public static void main(String[ ] args)
{
    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 over loading method resolution always takes care by compiler based on reference type. In overloading run object won't play any role

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 base on its requirement this process is called over riding.

The parent class method which is overridden is called overridden method and child class method which is overriding is called overriding method.

Example:

```
class Parent
{
    public void property()
    {
        System.out.println(" cash + land + gold");
    }
    public void marry()
    {
        System.out.println("subba laxmi");
    }
}

class Child extends Parent
{
    public void marry()
    {
        System.out.println("3sha | 4me | 9tara");
    }
}

Class Test
{
    Public static void main(String[ ] args)
    {
        1. Parent p = new Parent ( );
           p.marry();          →parent method

        2. Child c = new Child ( );
           c.marry();          → chid method

        3. Parent p1 = new Child ( );
           P1.marry();         → child method
    }
}
```

Diagram illustrating overriding:

- A red arrow labeled "overridden method" points from the `marry()` method in the `Parent` class to the `marry()` method in the `Child` class.
- A red arrow labeled "overriding" points from the `marry()` method in the `Child` class back to the `marry()` method in the `Parent` class.

*** 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 that is method signature must be same.
2. In overriding return type must be same but this rule is applicable until 1.4 version only from 1.5 version onwards we can take co-varient return types according to this child class method return type need not be same as parent method return type its child type also allowed.

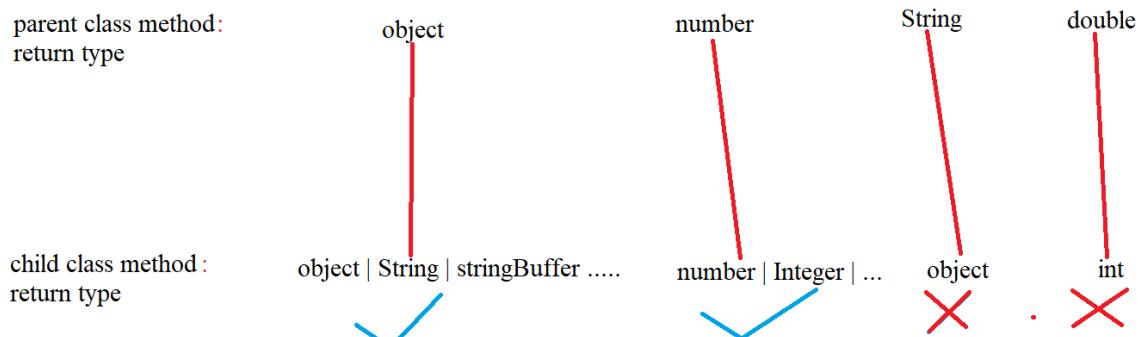
Example:

```
Class Parent
{
    Public Object m1()
    {
        Return null;
    }
}

Class child extends Parent
{
    Public String m1()
    {
        Return null;
    }
}
```

It is invalid in 1.4 verison
But form 1.5 version onwards it is valid

3. Co- variant concept is applicable only for object types but not for primitive types.



4. *** parent class private methods not available to the child and hence overriding concept not applicable for private methods.
5. Based on over requirement we can define exactly private method in child class it is valid but not overriding

Example:

it is valid but not overriding

```
Class Parent
```

```
{ Private void m1()
```

```
{ }
```

```
}
```

```
Class Child extends P
```

```
{ Private void m1()
```

```
{ }
```

```
}
```

g e

6. We can't override parent class final methods in child classes if we are trying to override we will get compile time error

Example:

```
Class Parent
{
    Public final void m1()
    {
    }
}

Class Child extends parent
{
    Public void m1();
}
```

//CE: m1() in Child cannot override m1() in Parent; overridden method is final

7. Parent class abstract method we should override in child class to provide implementation.

Example:

```
Abstract class Parent
{
    Public abstract void m1 ();
}

Class child extends Parent
{
    Public void m1 ();
    {
    }
}
```

8. We can override non abstract method as abstract

Example:

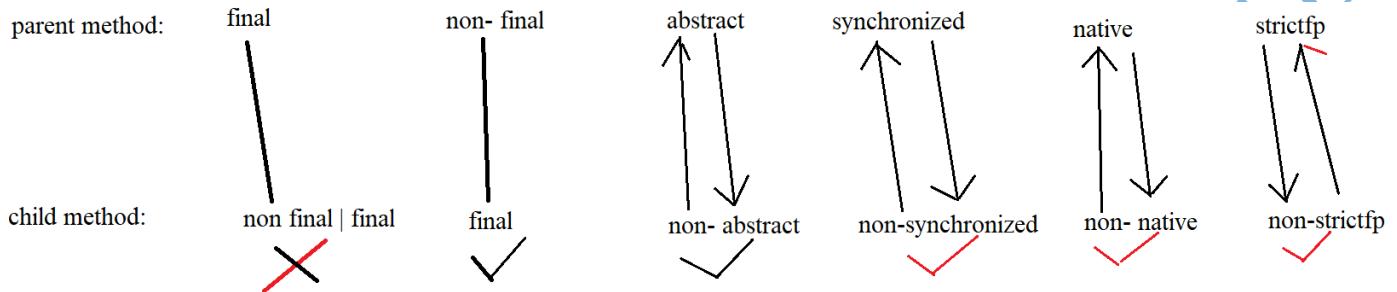
```
Class Parent
{
    Public void m1 ()
    {
    }
}

Abstract class Child extends Parent
{
    Public abstract void m1 ();
}
```

The main advantage of this approach we stop the availability of parent method implementation to the next level of child classes

9. In overriding the following modifiers won't keep any restriction

- a. Synchronized
- b. Native
- c. Strictfp



10. While overriding we cannot reduce scope of access modifier but we can increase the scope

Example:

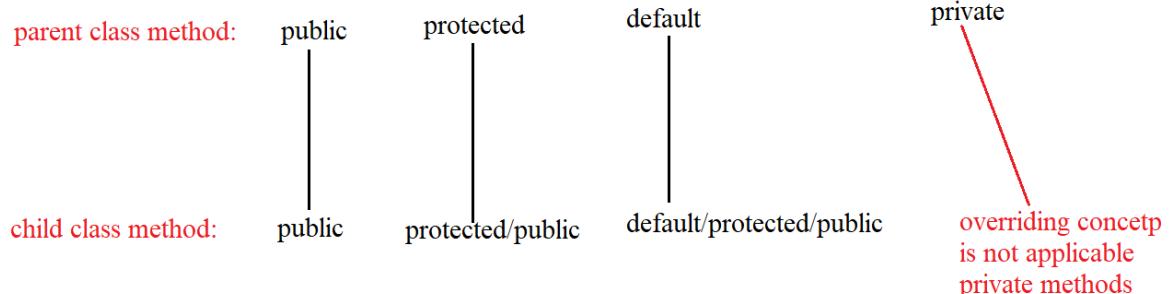
```
Class Parent
{
    Public void m1 ()
    {
    }
}
```

```
Class Child extend Parent
{
    Void m1 ()
    {
    }
}
```

CE: m1() in Child cannot override M1() in Parent; attempting to assign weaker access privileges ;was public

private < default < protected < public

private < default < protected < public



11. If child class method throws any checked exception compulsory parent class method should throw the same checked exception are it parent otherwise we will get compile time error but there are no restriction for unchecked exceptions.

Example:

```
Import java.io.*;
Class Parent
{
    Public void m1() throws IO Exception
    {
    }
}
Class Child extends Parent
{
    Public void m1() throws EOFException, InterruptedException
    {
    }
}(invalid)
```

//CE: m1() in Child cannot overridden m1() in parent; overridden method does not throw java.lang.InterruptedException

- i) P: public void m1() throws Exception
C: public void m1(); //valid
- ii) P:public void m1();
C: public void m1() throws Exception //invalid
- iii) P: public void m1() throws Exception
C: public void m1() throws IOException //valid
- iv) P: public void m1() throws IOException
C: public void m1() throws Exception //invalid
- v) P: public void m1() throws IOException
C: public void m1() throws FileNotFoundException,EOFException //valid
- vi) P: public void m1() throws IOException
C: public void m1() throws EOFException, interruptedException //invalid
- vii) P: public void m1() throws IOException
C: public void m1() throws AE,NPE,CCE //valid

Overriding with respect to static methods

1. We cannot override a static method as non static otherwise we will compile time error

Example:

```
Class Parent
{
    Public static void m1()
    {
    }
}

Class Child extends Parent
{
    Public void m1()
    {
    }
}

//CE: m1() in Child cannot override m1() in Parent; overridden method is static
```

2. Similarly we cannot override a non static method as static method

Example:

```
Class Parent
{
    Public void m1()
    {
    }
}

Class Child extends Parent
{
    Public static void m1()
    {
    }
}

//CE: m1() in Child cannot override m1() in Parent; overriding method is static
```

3. If both parent and child class methods are static then we won't get any compile time error it seems overriding concept applicable for static methods but it is not overriding and it is method hiding.

Example:

```
class Parent
{
    public static void m1()
    {
    }
}

class Child extends Parent
{
    public static void m1()
    {
    }
}
```

it is method hiding
but no overriding

Method hiding

All rules of method hiding are exactly same as overriding except the following differences

Method hiding	Overriding
1. both parent and child class method should be static	1. both parent and child class method should be non – static
2. compiler is responsible for method resolution based on reference type	2. JVM is responsible for method resolution base on runtime object
3. it is also known as compile time polymorphism or static polymorphism or early binding	3. it is also known as runtime polymorphism or dynamic polymorphism or late binding

Example;

Class Parent

```
{  
    public static void m1()  
    {  
        System.out.println("parent")  
    }  
}
```

class Child extends Parent

```
{  
    Public static void m1()  
    {  
        System.out.println("child");  
    }  
}
```

Class Test

```
{  
    Public static void main(String[ ] args)  
    {  
        Parent p = new Parent();  
        p.m1();      → parent  
  
        Child c = new Child();  
        c.m1();      → child  
  
        Parent p1 = new Child ();  
        p1.m1();     → parent  
    }  
}
```

It is method hiding but
not overriding

If both parent and child class methods are non static then it will become overriding in this case the output is

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.

Example:

Class Parent

```
{  
    Public void m1(int... x)  
    {  
        System.out.println("parent");  
    }  
}
```

Class Child extends Parent

```
{  
    Public voidm m1(int x)  
    {  
        System.out.println("child");  
    }  
}
```

Class test

```
{  
    Public static void main(String[ ] args)  
    {  
        Parent p = new Parent();  
        p.m1(10); → parent  
  
        Child c = new Child();  
        c.m1(10); → child  
  
        parent p1 = new Child();  
        p1.m1(10); → parent  
    }  
}
```

It is overloading but not overriding

In the above program if we replaced child method with var- arg method 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).

```
Class Parent
{
    Int x = 888;
}

Class Child extends Parent
{
    Int x = 999;
}

Class Test
{
    Public static void main(String[] args)
    {
        Parent p = new Parent();
        System.out.println(p.x);      →888

        Child c = new Child();
        System.out.println(c.x);      →999

        Parent p1 = new Child();
        System.out.println(p1.x);     →888
    }
}
```

P→ non-static C→non-static	P→static C→non- static	P→non-static C→non-static	P→static C→static
888	888	888	888
999	999	999	999
888	888	888	888

Difference between overloading and overriding:

Property	Overloading	Overriding
1. method names	Must be same	Must be same
2.argument types	Must be different (at least order)	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 version on wards co-varient return types also allowed
5. private, static and final methods	Can be overloaded	Cannot be overridden
6. access modifiers	No restriction	We cannot reduce scope of access modifier but we can increase the scope
7. throw clause	No restriction	If child class method throws any checked exception compulsory parent class method should throw the same checked exception or its parent but no restriction for unchecked exception
8.method resolution	Always take care by compiler based on reference type	Always takes care by JVM based on runtime object
9.it also known as	Compile time polymorphism or static polymorphism or early binding	Run time polymorphism or dynamic polymorphism or late binding

Note:

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.,

In overriding everything we have to check like method name, argument types, return types, access modifier, throws class etc.,

Consider the following method in parent class

Public void m1(int x) throws IOException

In the child class which of the following methods we can take

- | | |
|---|---|
| 1. Public void m1(int i); | → overriding(valid) |
| 2. Public static int m1(long l); | →overloading(valid) |
| 3. Public static void m1(int i); | → overriding(valid) |
| 4. Public void m1(int i) throws exception; | →overriding(valid) |
| 5. Public static abstract void m1(double d); | CE: illegal combination of modifiers |

Polymorphism

One name but multiple forms is the concept of polymorphism

Example 1:

Method name is the same but we can apply for different type of arguments (overloading)
Abs(int)
Abs(long) → overloading
Absl(float)

Example 2:

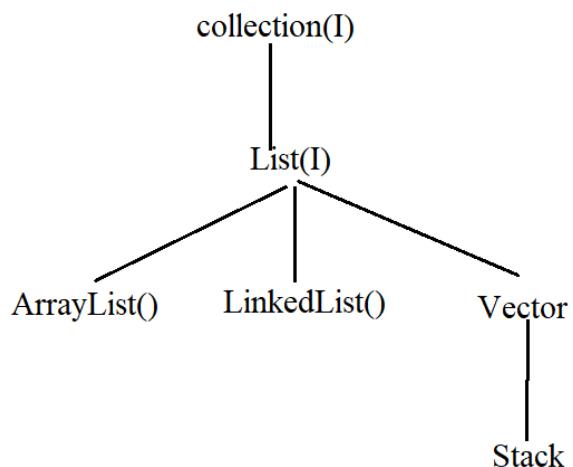
Method signature is same but in parent class one type of implementation and in the child class another type of implementation (overriding)

```
Class Parent
{
    Public void marry()
    {
        System.out.println("subbalaxmi");
    }
}
Class Child extends Parent
{
    Public void marry()
    {
        System.out.println("3sha | 9tara|4me");
    }
}
```

Example 3:

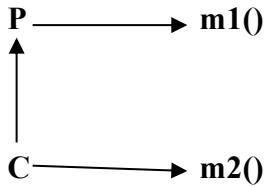
Usage of parent reference to hold child object is the concept polymorphism

List 1 =
 |→ new ArrayList();
 |→ new LinkedList();
 |→ new Stack();
 |→ new Vector();



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 cannot call child specific methods

Parent p = new Child();
p.m1(); → valid
p.m2(); → CE: cannot find symbol; symbol: method m2(); location: class P



But by using child object reference we can call both parent and child class methods

Child c = new Child();
c.m1(); → valid
c.m2(); → valid

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 example the first element present in the array list can be any type It may be student object, customer object, string object or string buffer object hence the return type of get method is object which can hold any object

Object o = l.get(0);

Child c = new Child();	Parent p = new Child();
Eg: ArrayList l = new ArrayList();	Eg: List l = new ArrayList();
1. we can use this approach if we 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 and child class methods(this is the advantage of this approach)	2. by using parent reference we can call only methods available in parent class and we cannot call child specific methods(this is the disadvantage this approach)
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)

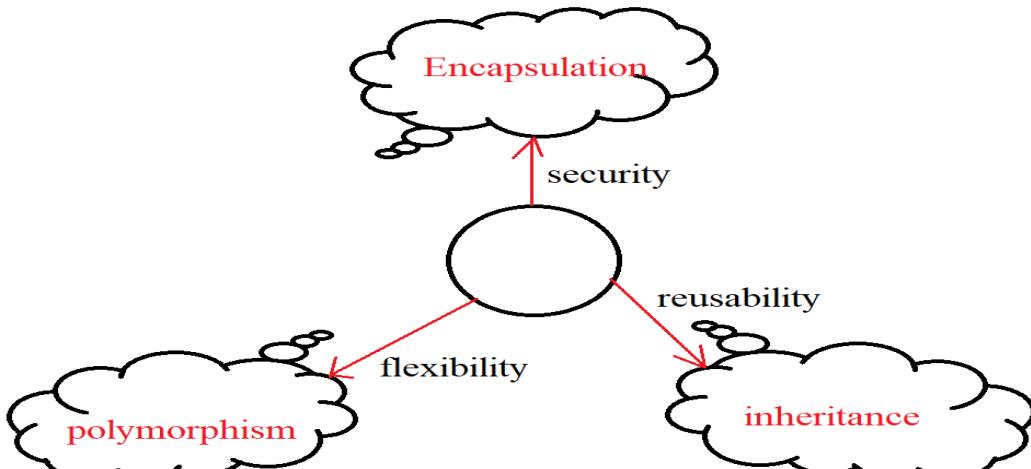
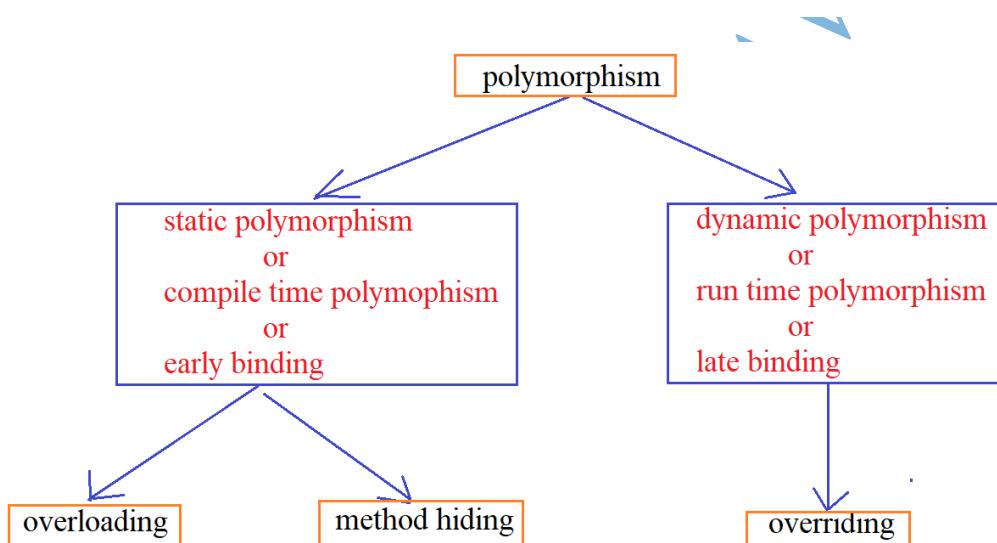


fig: 3 pillars of oops



Beautiful definition of polymorphism:

A BOY starts LOVE with the word FRIENDSHIP, but GIRL ends LOVE with the same word FIRENDSHIP. Word is the same but attitude is different. This beautiful concept of OOPS is nothing but polymorphism.....

Coupling

The degree of dependence between the components is called coupling.

If dependence is more than it is considered as **tightly coupling** and dependence is less then it is considered as **loosely coupling**

Example:

```
Class A
{
    Static int I = B.j
}
```

```
Class B
{
    Static int j = cC.k
}
```

```
Class C
{
    Static int k = D.m1();
}

Class D
{
    Public static int m1()
    {
        Return 0;
    }
}
```

The above components are said to be tightly coupled with each other because dependency between the components is more

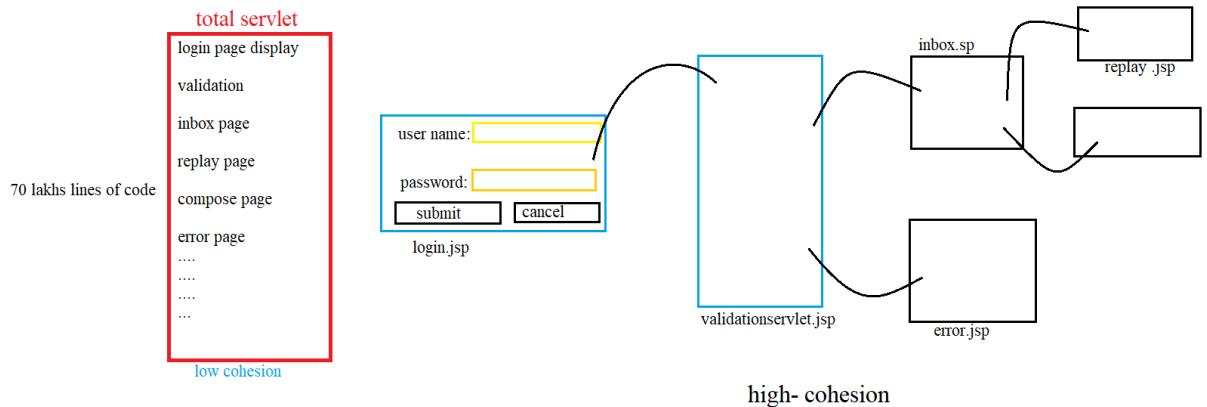
Tightly coupling is not a good programming practice because it has several serious disadvantages.

1. Without effect remaining components we cannot modify any component and hence enhancement will be difficult
2. It suppress reusability
3. It reduces maintainability of the application.

Hence we have to maintain dependence between the components as less as possible that is loosely coupling is a good programming practice

cohesion

For every component a clear well defined functionality is defined then that component is said to be fallow **high cohesion**



For all components are written into single component is called low cohesion hence it is not recommended to write all components into a single component

High cohesion is always good programming practice because it has several advantages

1. Without effecting remaining components we can modify any component hence enhancement will become easy.
2. It promotes reusability of the code (where ever validation is required we can reuse the same validate servlet 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

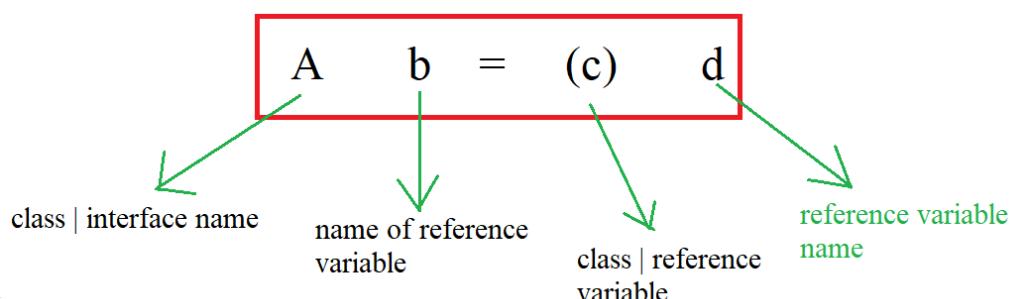
Example:

Object o = new String("durga"); → valid

We can use interface reference to hold implemented class object.

Example:

Runnable r = new Thread(); → valid



Mantra -1: (compile time checking one)

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 compile time error saying

“Inconvertible types

Found: d type

Required: c”

Example 1:

Object o = new String (“malach”);
StringBuffer sb = new (StringBuffer) o;
→ Valid

Mantra-2 (compile time checking two):

‘c’ must be either same or derived type of ‘A’ otherwise we will get compile time error saying “incompatible type; found: c; required:A

Example 1:

Object o = new String (“malachi”);
StringBuffer sb = new (StringBuffer) o;
→ Valid

Mantra-3 (run time checking two):

Run time object type of ‘d’ must be either same or derived type of ‘c’ otherwise we will get runtime exception saying “ClassCastException”

Example 1:

Object o = new String (“malachi”);
StringBuffer sb = new (StringBuffer) o;

→invalid : RE: ClassCastException:
java.lang.String cannto be cast to
java.lang.StringBuffer.

Example 2:

String s = new String(“malachi”);
StringBuffer sb = (String Buffer)s;

→Invalid: CE incompatible types;
found: java.lang.String;
required: java.lang.StringBuffer

Example 2:

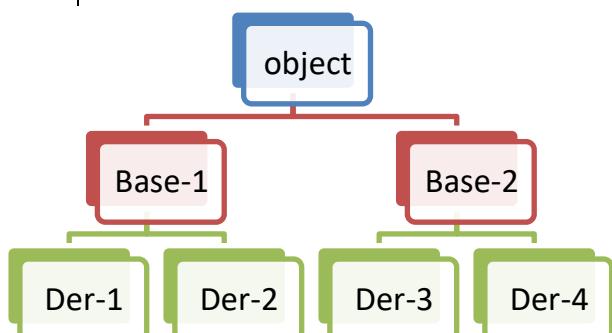
Object o = new String(“malachi”);
StringBuffer sb = (String) o;

→Invalid: CE incompatible types;
found: java.lang.String;
required: java.lang.StringBuffer

Example 2:

Object o = new String(“malachi”);
Object o1 = (String) o;

→valid



Example:

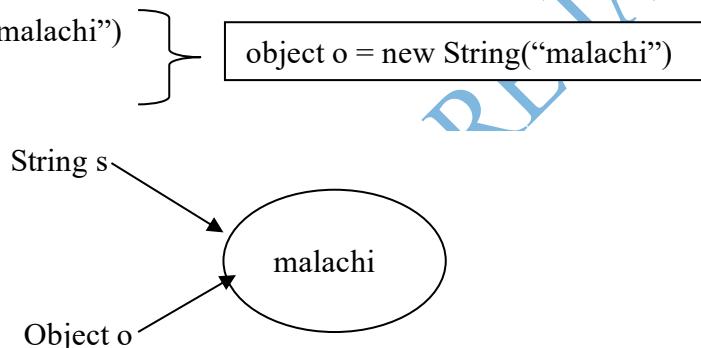
base2 b = new der4();

1. Object o = (base2)b; //valid
2. Object o = (base1)b // CE: incompatible types; found: base2; required: base1
3. Object o = (der3)b; //RE: ClassCastException
4. Base2 b1 = (base1)b; // CE: incompatible types; found: base2; required: base1
5. Base1 b1 = (der4) b; //CE: incompatible types; found: der4; required: base1
6. Base1 b1 = (der1)b; // CE: incompatible types; found: base2; required: der1

Strictly speaking through typecasting we are not creating any new object. For the existing object we are providing another type of reference variable that is we are performing type casting (conversion) but not object casting.

Example:

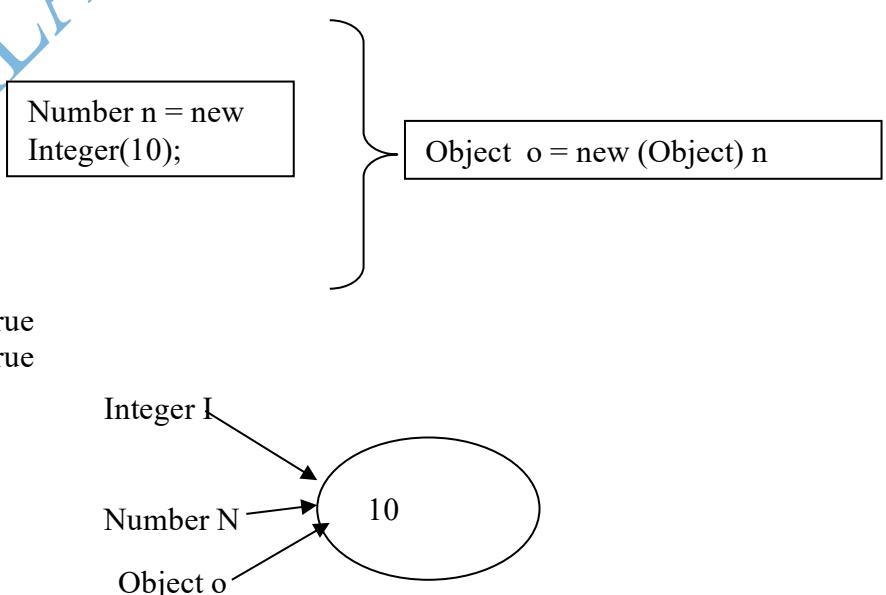
String s = new String("malachi")
Object o = (Object)s;



Example 2:

Integer I = new Integer(10);
Number n = (Number) I;
Object o = (Object) n;

System.out.println(I == n); //true
System.out.println(n == o); //true



Note:

C c = new C();

(B)c; → B b = new C();

(A)((B)c); → A a = new C()



Example:

Child c = new Child();

c.m1(); //valid

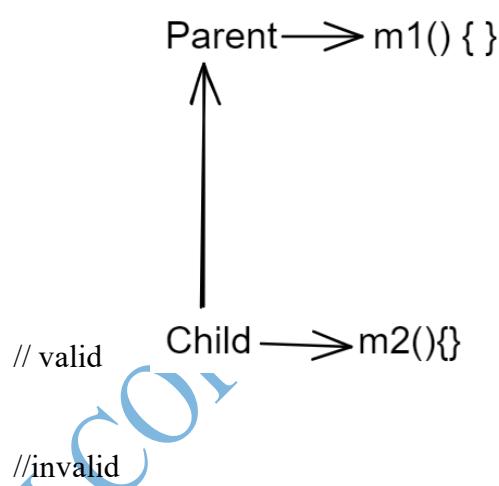
c.m2(); //valid

((p)c).m1(); → Parent p = new Child();
p.m1(); // valid

((p)c).m2(); → Parent p = new Child();
p.m2(); //invalid

REASON:

Parent reference can be used to hold child object but by using that reference we cannot call child specific methods and we can call only the methods available in parent class.



Example 2:

C c = new C();

c.m1();

((B)c).m1();

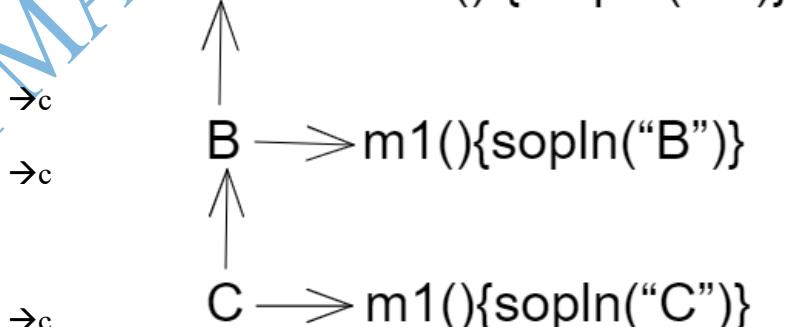
B b = new C();

b.m1()

((A)((B)c).m1());

A a = new C();

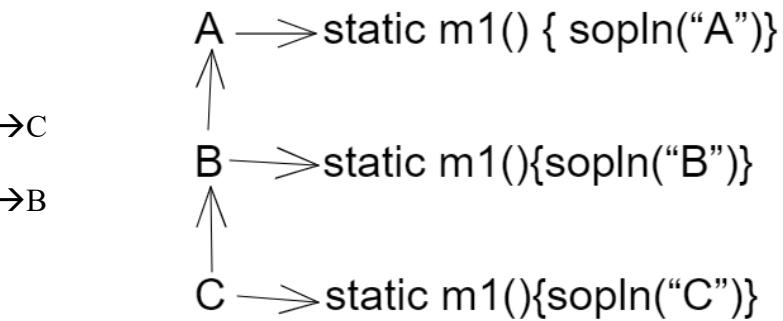
a.m1();



Reason: It is overriding, and method resolution is always based on runtime object type

Example;
C c = new C();

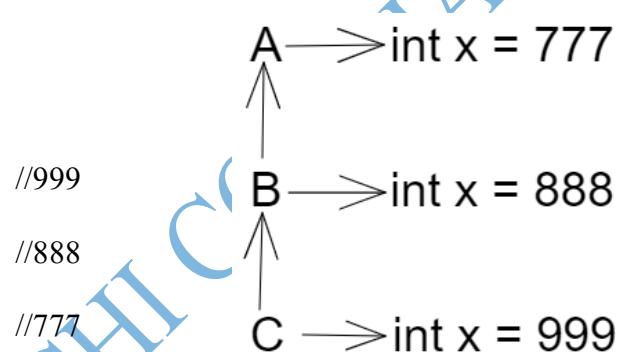
c.m1();
((B)c).m1();
B b = new C();
b.m1()
((A)((B)c).m1());
A a = new C();
a.m1();



Reason: it is method hiding and method and method resolution is always based on reference type

Example :

C c = new C();
System.out.println(c.x);
System.out.println(((B)c)).x);
System.out.println((A((B)c)).x);

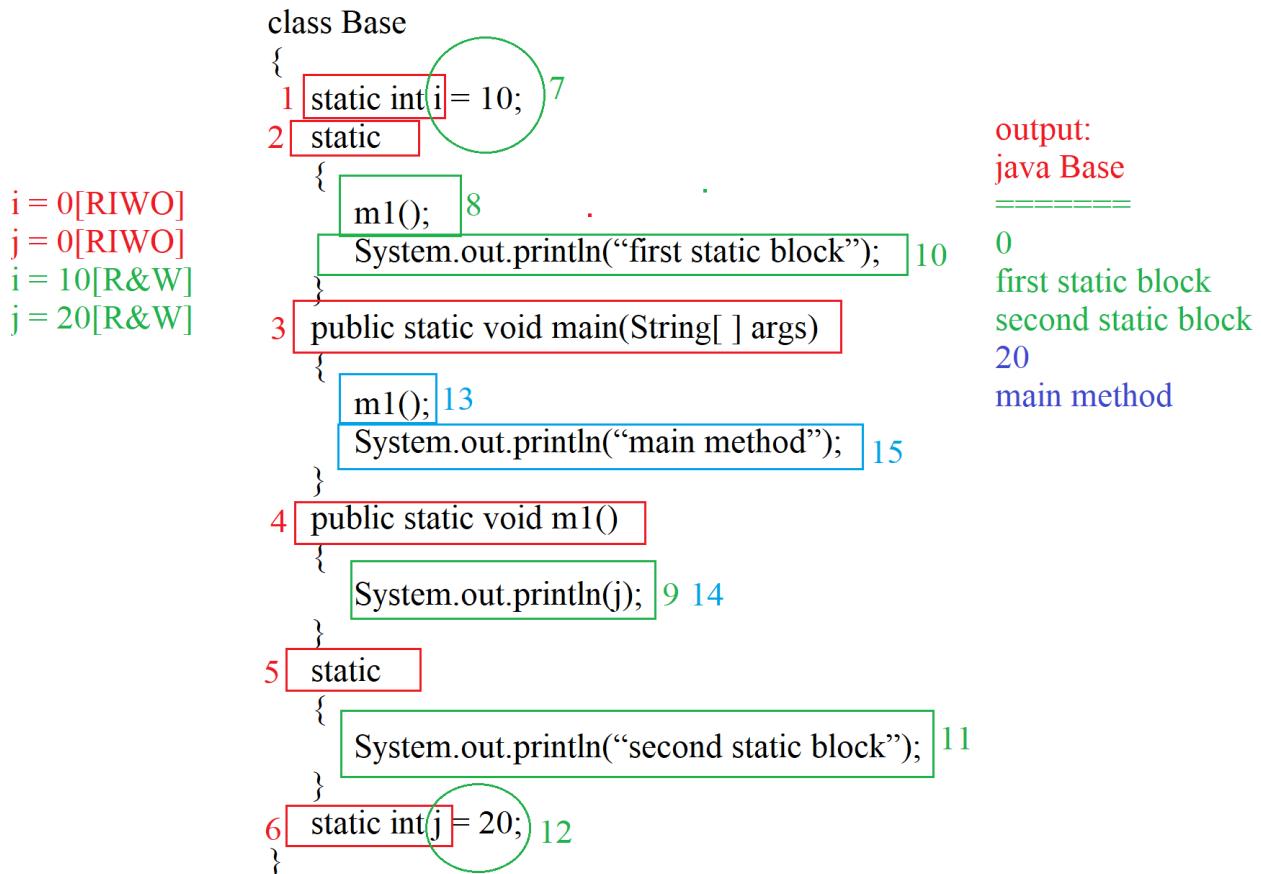


Reason: variable resolution is always based on reference type but not based on runtime object

Static control flow

Whenever we are executing a java class the following sequence of steps will be executed as the part of 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 methods (13 to 15)



Read indirectly write only (RIWO)

Inside static block we are trying to read a variable that read operation is called **direct read**

If we are calling a method and within that method trying to read a variable that read operation is called **indirect read**

example:

```
class Test{
    Static int I =10;      → I = 0[RIWO]
    Static{
        M1();
        System.out.println(i); → direct read
    }
```

```

Public static void m1(){
    System.out.println(i); → indirect read
}
}

```

If a variable is just identified by the JVM and original value not assigned then the variable is said to be **in read indirectly and write only state(RIWO)**

If a variable is in read indirectly and write only state(RIWO) then we cannot perform direct read but we can perform indirect read.

If we are trying to read directly then we will get compile time error saying “illegal forward reference”

```

Class Test
{
    1.Static int x = 10;
    2.Static
    {
        System.out.println(x);
    }
}

```

o/p: 10
RE: NoSuchMethodError:
main

```

Class Test
{
    1.Static
    {
        System.out.println(x);
    }
    2.Static int x = 10;
}

```

X = 0[RIWO]
CE: illegal forward reference

```

class Test
{
    1.Static
    {
        m1()
    }
    2.Public static void
m1()
{
    System.out.println(x);
}
    3.Static int x = 10;
}

```

o/p:
0
RE: NoSuchMethodError:
main

Static block

Static block will be execute at the time class loading hence at the time of class loading if we want perform any activity we have to define that inside static block.

Example 1:

At the time of java class loading the corresponding native library should be loaded hence we have to define this activity inside static block

```

Class Test{
    Static{
        System.loadLibrary("native library path");
    }
}

```

Example 2:

After loading every data base driver class we have to register driver class with driver manager but inside data base driver class there is a static block to perform this activity and we are not responsible to register explicitly

```
Class DbDriver
{
    Static
    {
        Register this driver with driver manager;
    }
}
```

Note: within a class we can declare any number of static blocks but all this static blocks will be executed from top to bottom.

Without writing main method is it possible to print some statements to the console?

Yes, by using static block 1.6 version from 1.7 we cannot print without main method

```
Class Test
{
    Static
    {
        System.out.println("hello I can Print");
        System.exit(0);
    }
}
o/p: hello I can Print
```

Without writing main method and static block is it possible to print some statements to the console?

Yes, off course there are multiple ways it is applicable for 1.6 version from 1.7 version it is not applicable

```
Class Test
{
    Static int x = m1();
    Public static int m1()
    {
        System.out.println("he
ll I can print");
        System.exit(0);
        Return 10;
    }
}
```

```
Class Test
{
    Static Test t = new
Test();
{
    System.out.println("he
ll I can print");
    System.exit(0);
}
```

```
Class Test
{
    Static Test t = new
Test();
{
    System.out.println("he
ll I can print");
    System.exit(0);
}
```

Note:

From 1.7 version onwards main method is mandatory to start program execution hence from 1.7 versions onwards without writing main method it is impossible to print some statement to the console

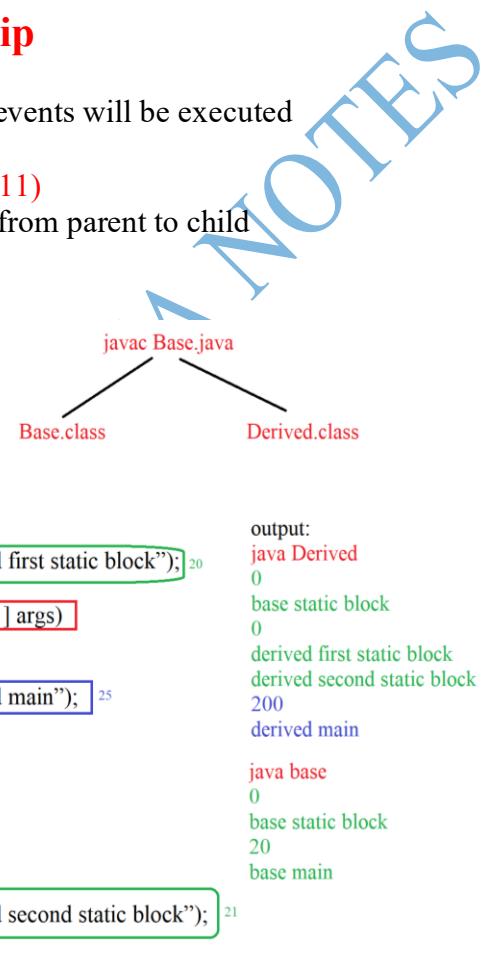
Static control flow in parent to child relationship

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 (1 to 11)
2. Execution of static variable assignments and static blocks from parent to child (12 to 22)
3. Execution of only child class main method(23 to 25)

```
class Base
{
    1 static int i = 10; 12
    2 static
    {
        m1(); 13
        System.out.println("base static block"); 15
    }
    3 public static void main(String [ ] args)
    {
        m1();
        System.out.println("base main");
    }
    4 public static void m1()
    {
        System.out.println(j); 14
    }
    5 static int j = 20; 16
}
```

```
class Derived extends Base
{
    6 static int x = 100; 17
    7 static
    {
        m2(); 18
        System.out.println("derived first static block"); 20
    }
    8 public static void main(String[ ] args)
    {
        m2(); 23 ■
        System.out.println("derived main"); 25
    }
    9 public static void m2()
    {
        System.out.println(y); 19
    }
    10 static
    {
        System.out.println("derived second static block"); 21
    }
    11 static int y = 200; 22
}
```



Note:

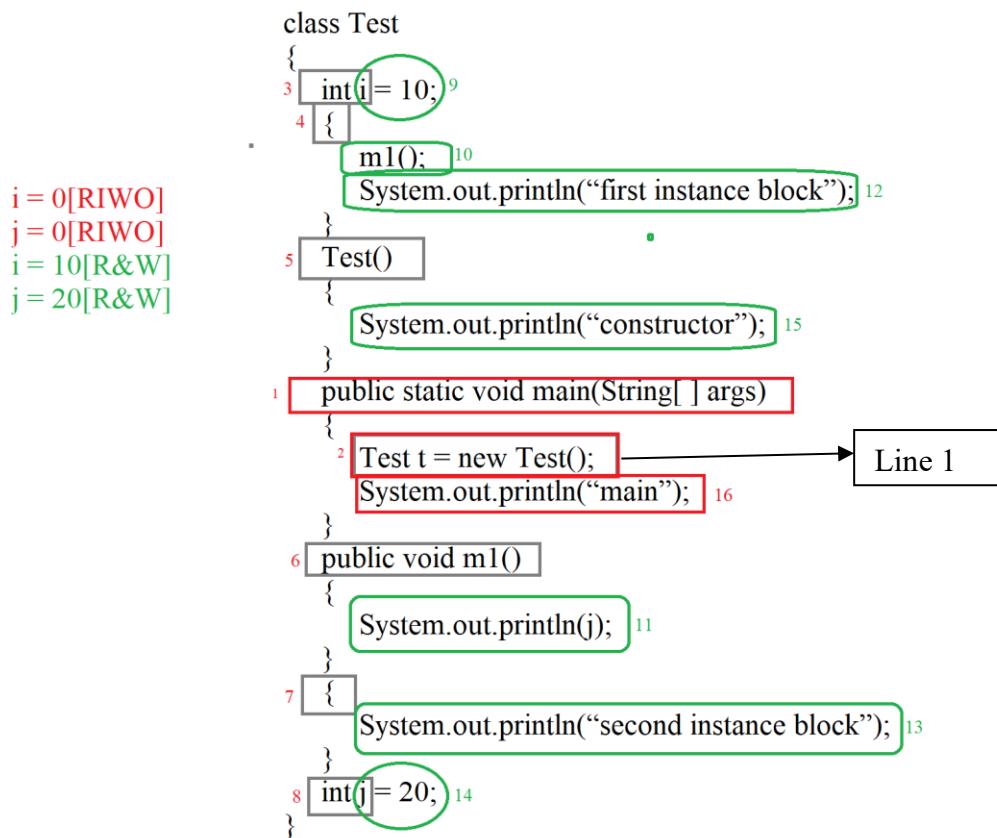
Whenever we are loading child class automatically parent class will be loaded but whenever we are loading parent class child class won't be loaded (because parent class members by default available to the child class whereas child class members by default won't available to the parent)

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 instance control flow

1. Identification of instance member 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)



If we comment line one then the output is “main”

Note:

1. static control flow is one time activity which will be performed at the time class loading.
But 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.

Instance control flow in parent to child relationship

When we are creating child class object the following sequence of events will be performed automatically as the part of instance control flow

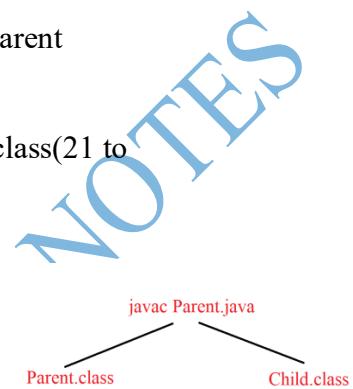
1. Identification of instance members form parent to child.(4 to 14)
2. Execution of instance variable assignments and instance blocks only in parent class(15 to 19).
3. Execution of parent constructor(20)
4. Execution of instance variable assignments and instance blocks in child class(21 to 26)
5. Execution of child constructor(27)

```

class Parent
{
    4 int i= 10; 15
    5 {
        m1(); 16
        System.out.println("parent instance block"); 18
    }
    6 Parent()
    {
        System.out.println("parent constructor"); 20
    }
    1 public static void main(String[ ] args)
    {
        Parent p = new Parent();
        System.out.println("parent main");
    }
    7 public void m1()
    {
        System.out.println(j); 17
    }
    8 int j= 20; 19
}
  
```

```

class Child extends Parent
{
    9 Int(x= 100; 21
    10 {
        M2(); 22
        System.out.println("child first instance block"); 24
    }
    11 Child()
    {
        System.out.println("child constructor"); 27
    }
    2 Public static void main(String[ ] args)
    {
        3 Child c = new Child();
        System.out.println("child main"); 28
    }
    12 Public void m2()
    {
        System.out.println(y); 23
    }
    13 {
        System.out.println("child second instance block"); 25
    }
    14 Int(y= 200; 26
}
  
```



```

java Child
0
parent instance block
parent constructor
0
child first instance block
child second instance block
child constructor
child main
  
```

Example 1:

```

Class Test
{
    {
        System.out.println("first instance block");
    }
    Static
    {
        System.out.println("first static block");
    }
    Test
    {
        System.out.println("constructor");
    }
    Public static void
    main(String[ ] args);
}
  
```

```

    Test();
    System.out.println("main");
    Test t1 = new
    Test();
    System.out.println("second static block");
    Test t2 = new
    Test();
    Static
    {
        System.out.println("second static block");
    }
    System.out.println("second instance block");
}
  
```

Output:

```

First static block
Second static block
First instance block
Second instance block
Constructor
Main
First instance block
Second instance block
Constructor
  
```

Example 2:

Public class Initialization

```
{
    Private static String m1(String msg)
    {
        System.out.println(msg);
        Return msg;
    }
    Public Initialization()
    {
        M = m1("1");
    }
    {
        M = m1("2");
    }
    String m = m1("3");
}
```

```

    Public static void main(String[ ] args)
    {
        Object o = new
initialization();
    }
}

Output:
2
3
1

M = null
2
3
1

```

Example3:

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
Initialization();
    }
}

Output:
1
3
2

M = null
4
3
2

```

Note:

*** from static area we cannot access instance members directly because while executing static area JVM may not identify instance members

Class Test{

Int x = 10

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

→CE: non- static variable x cannot be referenced from a static context

***** In how many ways we can create an object in java or in how many ways we can get object in java?**

By using new operator

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

By using newInstance() method:

```
Test t = new (Test)Class.forName("Test").newInstance();
```

By using factory method:

```
Runtime r = Runtime.getRuntime();
DateFormat df = DateFormat.getInstance();
```

By using clone() method:

```
Test t1 = new Test();
Test t2 = (Test)t1.clone();
```

By using deserialization:

```
FileInputStream fis = new FileInputStream("abc.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
Dog d2 = (Dog)ois.readObject();
```

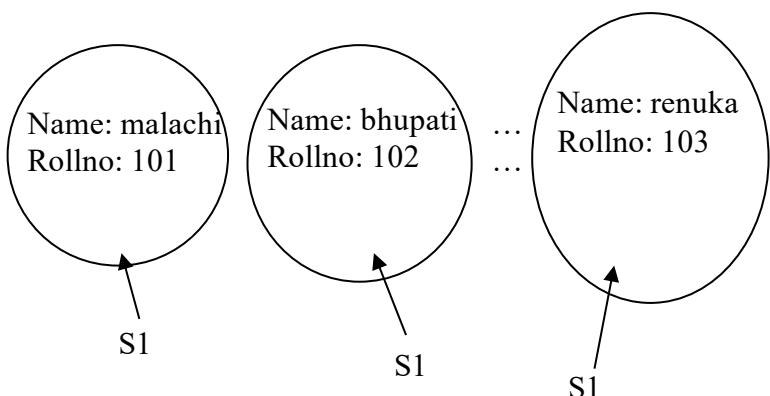
Constructor

Once we create an object compulsory, we should perform initialization then only the object is in 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 the constructor is to perform initialization of an object**

```
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("malachi", 101);
        Student s2 = new Student("bhupathi",102);
    }
}
```

Note: The main purpose of constructor is to perform initialization of an object but not to create object



Difference between constructor and instance block

The main purpose of constructor is to perform initialization of an object.

But other than 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 data base for every object creation or incrementing count value for every object creation etc.,)

Both constructor and instance block have their own different purposes and replacing one concept with another concept may not work always.

Both constructor and instance block will be executed for every object creation but instance block first followed constructor.

Demo program to print number of objects created for a class

```
Class Test
{
    Static int count = 0;
    {
        Count++;
    }
    Test()
    {
    }
    Test(int i)
    {
    }

    Test (double d)
    {
    }

    Public static void main(String[ ] args)
    {
        Test t1 = new Test();
        Test t2 = new Test(10);
        Test t3 = new Test(10.5);
        System.out.println*: the no of
        objects created:"+ count);
    }
}
```

Rules of writing constructors:

1. Name of the class and name of the constructor must be matched.
2. Return type concept not applicable for constructor even void also.
3. By mistake if we are trying to declare return type for the constructor we won't get any compile time error because compiler treats it as a method.

```
Class Test
{
    Void Test() // it is a method but not consturctor
    {
    }
}
```

4. Hence it legal (but stupid) to have a method whose name is exactly same as class name.

```
Class Test
{
    Void Test()
    {
        System.out.println("it is method but not a constructor");
    }
    Public static void main(String[ ] arg)
    {
        Test t = new Test();
        t.Test();
    }
}
```

5. *** the only applicable modifiers for constructors are public, private, protected and default. If we are trying to use any other modifiers we will get compile time error

```
Class Test
{
    Static Test()
    {
        } // CE: modifier static not allowed here
}
```

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 that is if we are writing at least one constructor then compiler won't generate default constructor 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 access modifier of class(this rule is applicable only for public and default).
3. It contains only one line “super();” it is a no argument call to super class constructor.

Programmers code	Compiler generated code
class Test{ }	class Test{ Test() { Super(); } }
public class Test{ }	public class Test{ public Test(){ super(); } }
public Class Test{ void Test(){ } }	public Class Test{ public Test(){ super(); } void Test() { } }
class Test{ Test(){ } }	class Test{ Test(){ Super(); } }
class Test{ Test(int i){ super(); } }	class Test{ Test(int i){ super(); } }
class Test{ Test(){ this(10); } Test (int i){ } }	class Test{ Test(){ this(10); } Test (int i){ super(); } }

The first line every constructor should be either super or this and if we are not writing anything then compiler will always place super()

Case 1:

We can take super() or this() only in first line of constructor. If we are trying to take anywhere else we get compile time error

```
Class Test
{
    Test()
    {
        System.out.println("constructor");
        Super();
    } // CE : call to super must be first statement in constructor
}
```

Case 2:

With in the constructor we can take either super() or this() but not both simultaneously

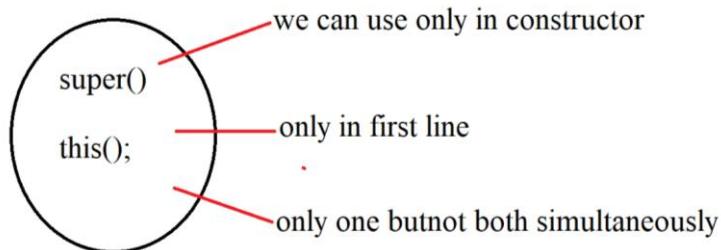
```
Calss Test
{
    Test()
    {
        Super();
        This();
    } //CE: call to this must be first statement in constructor.
}
```

Case 3:

We can use super() or this() only inside constructor if we are trying to use outside of constructor we can get compile time error.

```
Class Test
{
    Public void m1()
    {
        Super();
        System.out.println("hello");
    } //CE: call to super must be first statement in constructor
}
```

That is we can call a constructor directly from another constructor only



Difference between super(), this() and normal super, this?

Super(), this()	Super, this
This are constructor calls to call super class and current class constructors	This are keyword to refer super class and current class instance members
We can use only in constructors as first line	We can use anywhere except static area
We can use only once in a constructor	We can use any number of times.

Class Test

```

{
    Public static void main(String[ ] args)
    {
        System.out.println(super.hashCode());
    }
} // CE: non-static variable super cannot be referenced from a static context.

```

Overloaded constructor

Within a class we can declare multiple constructor and all this constructor having same name but different type of arguments hence all this constructor considered as overloaded constructors hence overloading concept applicable for constructor.

Class Test

```

{
    Test()
    {
        System.out.println("no-arg");
    }
    Test(int i)
    {
        This(10.5);
        System.out.println("int - arg");
    }
    Test(double d)
    {
        System.out.println("double - arg");
    }
}

```

Over loaded
constructor

```

Public static void main(String[ ] arg)
{
    Test t1 = new Test(); →double arg
                           →int – arg
                           →no- arg

    Test t2 = new Test(10); →double - arg
                           →int – arg

    Test t3 = new Test(10.5); →double - arg

    Test t4 = new Test(10l); → double - arg
}

```

For constructors' inheritance and overriding concepts are not applicable but overloading concept is applicable.

Every class in java including abstract class can contain constructor but interface cannot contain constructor.

Class Test	Abstract class Test	Interface
{ Test() } } }→valid	{ Test() } } }→valid	{ Test () } } }→ invalid

Case 1:

Recursive method call is a runtime exception saying stack overflow error

But in our program if there is chance of recursive constructor invocation then the code won't compile and we get compile time error

Class Test	M1()	Class Test
{ Public static void m10() { M2(); } Public static void m20() { M1(); } Public static void main(String[] args) { M1(); System.out.println("hello"); } }	M2() M1() Main()	{ Test() { This(10); } Test(int i) { This(); } Public static void main(String[] args) { System.out.println("hello"); } }

} →RE: Stackoverflow
} →CE: recursive constructor invocation

Case 2:

Class P	Class P	Class P
{ //P() {Super();} } Class C extends P { //C() {Super();} } }→ valid	{ P() { //Super();} } Class C extends P { //C() {Super();} } }→valid	{ P(int i) { //Super();} } Class C extends P { //C() {Super();} } }→invalid CE: cannot find symbol; symbol: constructor p(); location :classp

Note:

1. If parent class contains any argument constructor then while writing child classes we have to take special care with respect to constructor.
2. **** when ever writing any argument constructor it is highly recommended to write no-arg constructor

Case 3:

Class Parent

```
{
    Parent() throws IOException
    {

    }

    Class Child extends Parent
    {
        ...
    }
}
```

CE: unreported exception java.io.IOException
in default constructor

Class Parent

```
{
    Parent() throws IOException
    {

    }

    Class Child extends Parent
    {
        C() throws IOException | Exception | throwable
        {
            Super()
        }
    }
}
```

} → valid

If parent class constructor throws any checked exception compulsory child class constructor should throw the same checked exception or its parent otherwise the code won't compile.

Which of the following is valid?

- | | |
|---|-----------|
| 1. The main purpose of constructor is to create an object. | → invalid |
| 2. The main purpose of constructor is to perform initialization of an object. | → valid |
| 3. The name of the constructor need not be same as class name. | → invalid |
| 4. Return type concept applicable for constructors but only void | → invalid |
| 5. We can apply any modifier constructor | → invalid |
| 6. Default constructor generated by JVM | → false |
| 7. Compiler is responsible to generate default constructor | → true |
| 8. Compiler will always generate default constructor | → false |
| 9. If we are not writing no-arg constructor then compiler will generate default constructor | → invalid |
| 10. Every no argument constructor is always default constructor | → false |
| 11. Default constructor always is no-arg constructor | → true |
| 12. The first line inside a constructor either super() or this() if we are not writing anything then compiler will generate this() | → false |
| 13. For constructors both overloading and overriding concepts are applicable | → false |
| 14. For constructor inheritance concept applicable but not overriding | → false |
| 15. Only concrete class can contain constructor but abstract class cannot | → invalid |
| 16. Interface can contain constructors | → invalid |
| 17. Recursive constructor invocation is a runtime exception | → invalid |
| 18. If parent class constructor throws some checked exception then compulsory child class constructor should throw the same checked exception are its child | → false |

Singleton class

For any java class if we are allowed to create only one object such type of class is called singleton class

Example:

Runtime, Business Delegate, Service Locator

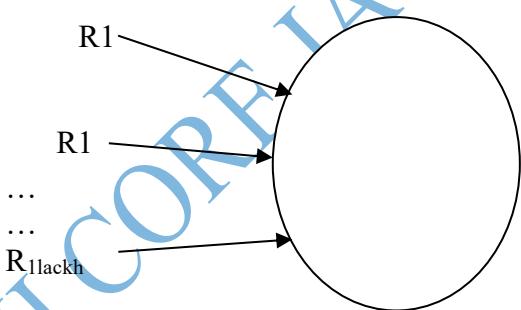
Advantage of singleton class:

If several people have same requirement then it is not recommended to create 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 and memory utilization will be improved

This is the central idea of singleton classes

```
Runtime r1 = Runtime.getRuntime();
Runtime r2 = Runtime.getRuntime();
.....
.....
.....
.....
Runtime r1lakhs = Runtime.getRuntime();
```



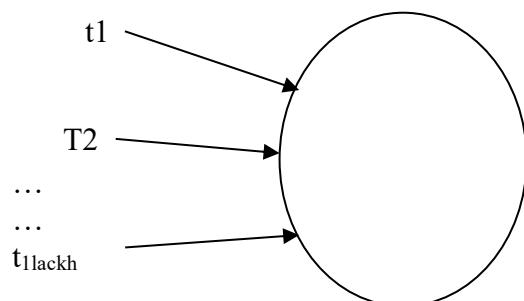
How to create our own singleton class?

We can create our own singleton classes for this we have to use private constructor and Private static variable and public factory method

Approach 1:

```
Class Test
{
    Private static Test t = new Test();
    Private Test()
    {
    }
    Public static Test getTest()
    {
        Return t;
    }
}
```

```
Test r1 = Test.getTest()
test t2 = test .getTest()
.....
Test t1lakhs = Test.getTest();
```



Note: runtime class is internally implemented by using this approach

Approach 2:

```
Class Test
{
    Private static Test t = null;
    Private Test()
    {
    }
    Public static Test getTest()
    {
        If (t == null)
        {
            T = new Test();
        }
        Return t;
    }
}
```

At any point of time for test class we can create only object hence test class is singleton class

Class is not final but we are not allowed to create child class how it is possible?

By declaring every constructor as private we can restrict child class creation

```
Class Parent
{
    Private Parent
    {
    }
}
```

For the above class it is impossible to create the child class

Exception handling

1. Introduction
2. Runtime stack mechanism
3. Default exception handling in java
4. Exception hierarchy
5. Customized exception handling b 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. Difference between final, finally, 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. Exception handling keywords summary
17. Various possible compile time errors in exception handling
18. Customized or user defined exception
19. Top -10 exceptions
20. 1.7 version enhancements
 - i. Try with resources
 - ii. Multi – catch block
 - iii.

Introduction:

An unexpected unwanted event that disturbs normal flow of the program is called exception

Example: tier punctured exception, sleeping exception, and file not found exception

It is highly recommended to handle exception and the main objective of exception handling is grace full termination of the program

Exception handling doesn't mean repairing an exception we have to provide alternative way to continue rest of the program normally is the concept of exception handling

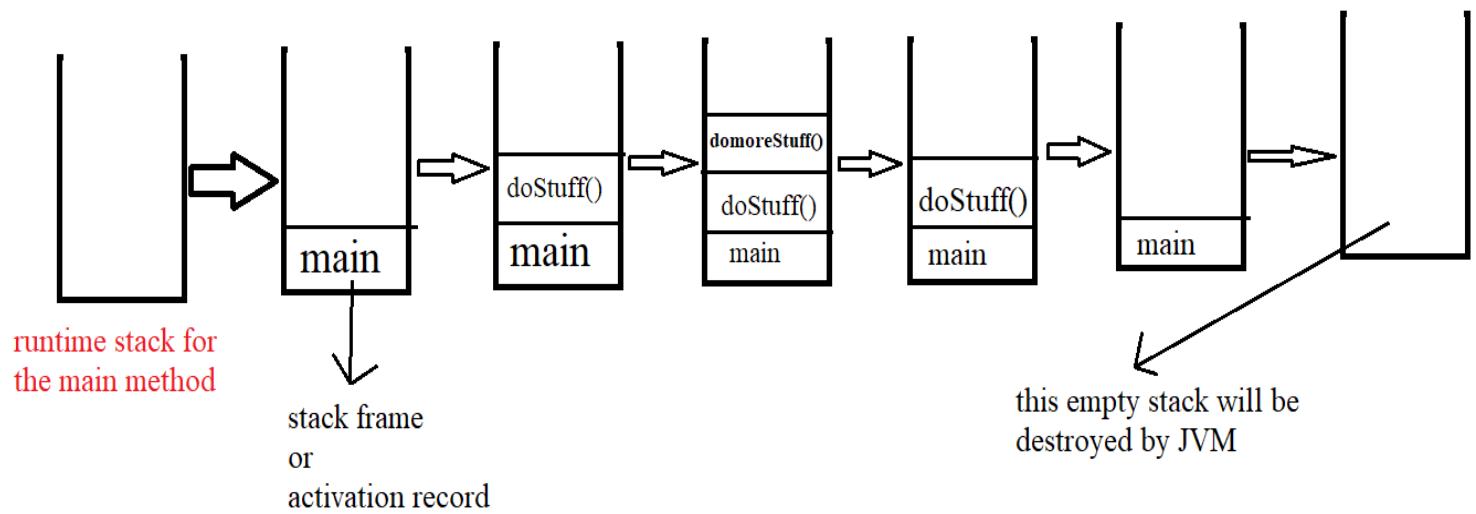
For example our programming requirement is to read data from remote file locating at London at runtime if London file is not available our program should not be terminated unnormally we have to provide some local file to continue rest of the program normally this way of defining alternative is nothing but exception handling.

```
Try
{
    Read data from remote file locating at london
}
Catch(FileNotFoundException e)
{
    Use local file and continue rest of the program normally
}
```

Runtime stack mechanism

For every thread JVM will create a runtime stack. Each and every method call performed by that thread will be stored in the corresponding stack each entry in the stack is called stack frame or activation record. After completing every method call the corresponding entry from the stack will be removed. After completing all method calls the stack becomes empty and that empty stack will be destroyed by JVM just before terminating the thread.

```
Class Test
{
    Public static void main(String[ ] args)
    {
        doStuff();
    }
    Public static void doStuff()
    {
        domoreStuff(0);
    }
    Publilc static void domoreStuff()
    {
        System.out.println("hello");
    }
}
output: hello
```



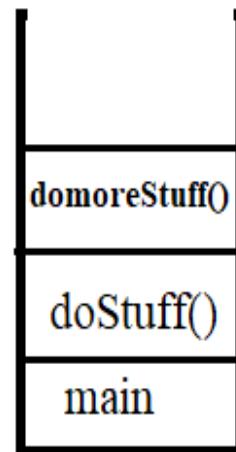
Default exception handling in java:

1. Inside a method if any exception occurs the method in which it is raised is responsible to create exception object by including the following information.
 - a. Name of exception
 - b. Description of exception
 - c. Location at which exception occurs (stack trace)
2. After creating exception object method handovers that object to the JVM .
3. JVM will check whether the method contains any exception handling code or not if the method doesn't contain exception code then JVM terminates that method abnormally and removes corresponding entry from the stack.
4. Then JVM identifies caller Method and checks whether the caller method contains any handling code are not if the caller method doesn't contain handling code then JVM terminates that caller method also abnormally and removes the corresponding entry from the stack
5. This process will be continued until main method and if the main method also doesn't contain handling code then JVM terminates main method also abnormally and removes corresponding entry from the stack.
6. Then JVM handovers responsibility of exception handling to default exception handler, which is the part of JVM.
7. Default exception handler prints exception information in the following format and terminates program abnormally.

**Exception in thread “main” name of the exception: description
Stack trace**

Example:

```
Class Test
{
    Public static void main(String[] args)
    {
        doStuff();
    }
    Public static void doStuff()
    {
        doMoreStuff();
    }
    Public static void doMoreStuff()
    {
        System.out.println(10/0);
    }
}
```

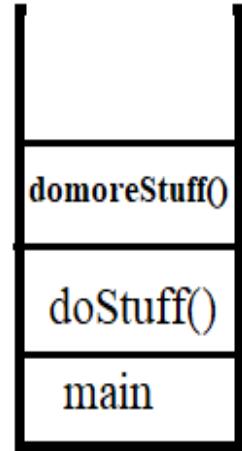


Output:

```
Exception in thread “main” java.lang.ArithmaticException: / by zero
At Test.doMoreStuff();
At Test.doStuff();
At Test.main();
```

Example2:

```
Class Test
{
    Public static void main(String[ ] args)
    {
        doStuff();
        system.out.println(10/0);
    }
    Public static void doStuff()
    {
        doMoreStuff();
        System.out.println("Hi")
    }
    Public static void doMoreStuff()
    {
        System.out.println("Hello");
    }
}
```



Output:

Hello

Hi

Exception in thread “main” java.lang.ArithmaticException: / by zero

At Test.main();

Note:

1. In a program if at least one method terminates abnormally then the program termination is abnormal termination
2. If all methods terminated normally then only program termination is normal termination

Exception hierarchy

Throwable class acts as root for java exception hierarchy

Throwable class defines two child classes

1. Exception
2. Error

1. Exception:

Most of the time exception or caused by our program and these are recoverable for example if our program requirement is to read out data from remote file located at London at run time if remote is not available then we will get runtime exception saying “FileNotFoundException”.

If FileNotFoundException occurs we can provide local file and continue rest of the program normally

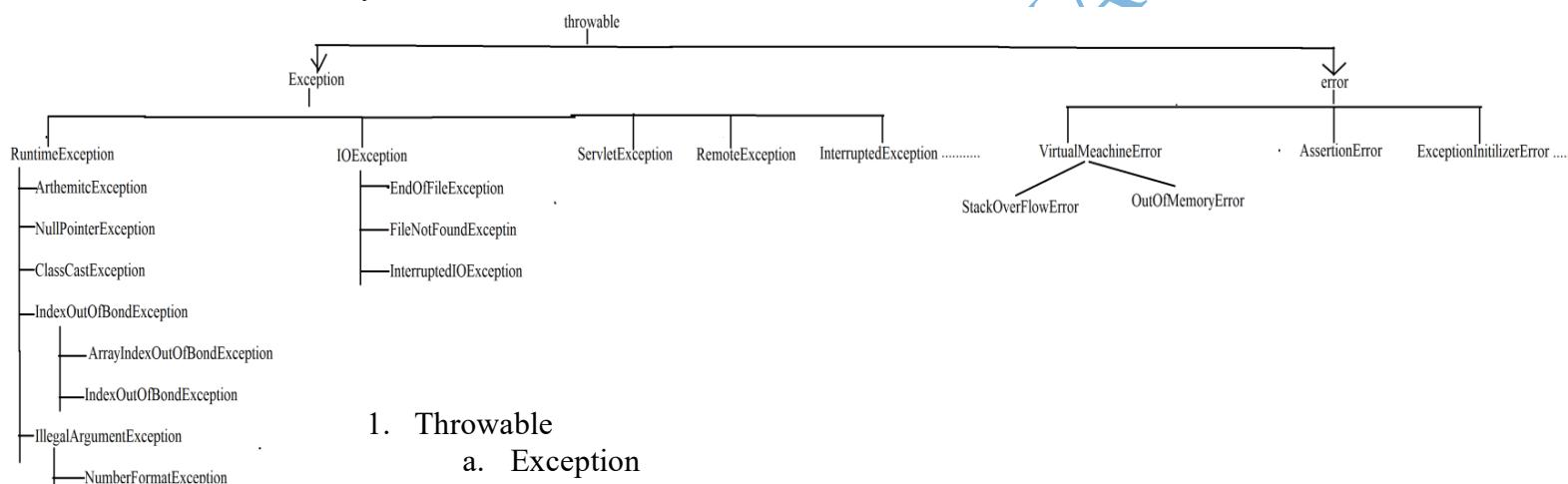
```

Try
{
    Read data from remote file locating at london
}
Catch(FileNotFoundException e)
{
    Use local file and continue read of the program normally;
}

```

2. error

Most of the times errors are not caused by our program and these are due to lack of system resources errors are not recoverable. For example if OutOfMemoryError occurs being a programmer we cannot do anything and the programme will be terminated abnormally. System admin or sever admin is responsible to increase heap memory.



1. Throwable

a. Exception

i. RuntimeException

1. ArithemticException
2. NullpointerException
3. ClassCastException
4. IndexOutOfBoundsException
 - a. ArrayIndexOutOFBoundException
 - b. IndexOutOfBoundsException
5. IllegalOutOfBondException

ii. IOException

1. EndOfFileException
2. FileNotFoundException
3. InterruptedIOException

iii. ServletException

iv. RemoteException

v. InterruptedException

b. Error

i. VirtualMeachineError

1. StackOverFlowError
2. OutOfMemoryError

ii. AssertionErroe

iii. ExceptionInIntializeError

***** Checked exception V/S unchecked exception

The exception which are checked by compiler for smooth execution of the program are called checked exception

Example:

HallTicketMissingException, PenNotWorkingException, FileNotFoundException etc.,

In our program if there is a chance of rising checked exception then compulsory we should handle that checked exception(either by try catch or throws keyword) otherwise we will get compile time error

The exception which are not checked by compiler whether programmer handling or not such type exception are called unchecked exception

Example:

ArthematicException, BombBlastException etc.,

Note:

1. Whether it checked or unchecked every exception occurs at runtime only there is no chance of occurring any exception at compile time
2. Runtime Exception and its child classes, error and its child classes are unchecked except this remaining are checked

Fully checked V/S partially checked

A checked exception is said to be fully checked if and only if all its child class also checked.

Example: IOException, InterruptedException.

A checked exception is said to be partially checked if and only if some of its child class are unchecked.

Example: Exception, throwable.

Note:

The only possible partially checked exception in java are

1. exception
2. Throwable

Describe the behaviour of following exceptions?

- | | |
|--|---|
| 1. IOException
→checked(fully checked) | 6. ArithmeticException →unchecked |
| 2. RuntimeException →unchecked | 7. NullPointerException →unchecked |
| 3. InterruptedException
→checked(fully checked) | 8. Exception
→ checked(partially checked) |
| 4. Error →unchecked | 9. FileNotFoundException
→checked(fully checked) |
| 5. Throwable
→ checked(partially checked) | |

Customized exception handling by using try, catch

It is highly recommended to handle exceptions

The code which may rise an exception is call risky code and we have to define that code inside try block and corresponding handling code we have to define inside catch block.

```
try
{
    Risky cod;
}
catch (Excetpiton e)
{
    Handling code;
}
```

Without try catch

```
class Test
{
    public static void main(String[ ] args)
    {
        System.out.println("statement 1");
        System.out.println(10/0);
        System.out.println("statement 3")
    }
}
```

Output:

Statement 1
RE: AE: /by zero

Abnormal termination

With try catch

```
class Test
{
    public static void main(String[ ] args)
    {
        System.out.println("statement 1");
        try
        {
            System.out.println(10/0);
        }
        catch (ArthmeticException E)
        {
            System.out.println(10/2);
        }
        System.out.println("statement 3")
    }
}
```

Output:

Statement 1
5
Statement 2
normal termination

Control flow in try catch

```
Try
{
    Statement 1;
    Statement 2;
    Statement 3;
}
Catch (X e)
{
    Statement 4;
}
Statement 5;
```

Case 1:

If there is no exception
1,2,3,5 normal termination

Case 2:

If an exception raised at statement 2 and corresponding catch block matched
1,4,5 normal termination

Case 3:

If an exception raised at statement 2 and corresponding catch block not matched
1, abnormal termination

Case 4:

If an exception raised at statement 4 or statement 5 then it is always abnormal termination

Note:

1. Within the try block if anywhere exception raised then rest of the try block won't be executed even though we handled that exception hence within the try block we have to take only risky code and length of try block should be as less as possible
2. In addition to try block there is a chance of raising an exception inside catch and finally blocks
3. If any statement which is not part of try block and rises an exception then it is always abnormal termination

Methods to print exception information

Throwable class defines the following methods to print exception information

Method	Printable format
printStackTrace()	Name of Exception: description stack trace
toString()	Name of exception: description
getMessage()	description

Class Test

```
{  
    Public static void main(String[ ] args)  
    {  
        Try  
        {  
            System.out.println(10/0);  
        }  
        Catch(ArithmeticException e)  
        {  
            e.printStackTrace(); → java.lang AE: / by zero ; at Test main ()  
            system.out.println(e); → java.lang.AE: / by zero  
            system.out.println(e.toString()); → java.lang.AE: / by zero  
            system.out.println(e.getMessage()); → / by zero  
        }  
    }  
}
```

Note:

Internally default exception handler will use printStackTrace() to print exception information to the console.

Try with multiple with catch blocks

The way of handling an exception is varied from exception to exception hence for every exception type it is highly recommended to take separate catch block that is try with multiple catch blocks is always possible and recommended to use

```
Try
{
    Risk code;
}
```

```
Catch (Exception e)
{
}
```

Worst programming practice

```
Try
{
    Risk code;
}

Catch (ArithmetException e)
{
    Perform alternative arithmetic
operation;
}

Catch (SQLException e)
{
    Use mysql db instead of oacle db;
}

Catch (FileNotFoundException e)
{
    Us local file instea of remote file;
}

Catch (Exception e)
{
    //default exception handling
}
```

Best approach programming practice

If try with multiple catch blocks present then the order of catch blocks is very important we have to take child first and then parent otherwise we will get compile time error saying

“Exception XXX has been already caught”

```
try
{
    Risky code;
}
catch (Exception e)
{
}
catch (arithmetidException e)
{ //CE: exception j.l.AE has already been
caught
}
}      → invalid
```

```
try
{
    Risky code;
}
catch (ArithmetException e)
{
}
catch (Exception e)
{
}
}      → valid
```

We cannot declare two catch blocks for the same exception other wise we will get compile time error

```
Try
{
    Risky code
}
Catch (ArthimeticException e)
{
}

}
Catch (ArthimeticException e)
{
}

}
Exception java.lang.ArthimeticException has already been caught
```

Final:

1. Final is the modifier applicable for classes, methods, and variables
2. If a class declared as final then we cannot extend that class that is we can not create child class for that class that is inheritance is not possible for final classes.
3. If a method is final then we can override that method in the child class.
4. If a variable declared as final we cannot perform reassignment for that variable.

Finally:

1. Finally is a block always associated with try catch to maintain clean up code

```
Try
{
    Risky code;
}
Catch(Excetption cod)
{
    Handling code;
}
Finally
{
    Cleanup code;
}
```

2. The speciality of finally block is it will be executed always irrespective of whether exception is raised or not raised and whether handled or not handled.

finalize():

finalize is a method always invoked by garbage collector just before destroying an object to perform clean up activities once finalize method completes immediately garbage collector destroyed that object.

Note:

Finally block is responsible to perform clean-up activities related to try block that is whatever resources we open as the part of try block will be closed inside a finally block

Where as finalize () method is responsible to perform clean-up activities related to object that is whatever resources associated with object will be deallocated before destroying an object by using finalize method.

Varies possible combinations of try – catch – finally

1. In try catch finally order is important
2. Whenever we are writing try compulsory we should write either catch or finally otherwise we will get compile time error that is try with catch or finally is invalid.
3. Whenever we are writing catch block compulsory try block must be required that is catch without try is invalid
4. Whenever we are writing finally block compulsory we should write try block that is finally without try is invalid
5. Inside try catch finally blocks we can declare try catch and finally blocks that is nesting of try catch finally is allowed.
6. For try catch finally blocks curly braces are mandatory.

1. Try
{
}
}
Catch(X e)
{
}

2. Try
{
}
Catch(X e)
{
}
Catch(Y e)

{
}

3. Try
{
}
}
Catch(X e)
{
}
Catch(X e)
{
}
CE: exception x has already been caught

```
4. Try
{
}
```

```
Catch(X e)
{
}
Finally
{
}
```

5. Try

```
{
}
Finally
{
}
```

6. Try

```
{
}
Catch(X e)
{
}
Try
{
}
Catch ()
{
}
```

7. Try

```
{
}
Catch(X e)
{
}
Try
{
}
Finally
{
}
```

8. Try
{
}
CE: try without catch or finally

9. Catch(X e)
{
}
CE: catch without try

10. Finally
{
}
CE: finally without try

11. Try
{
}
Finally()
{
}
Catch(X e)
{
}
CE: catch with out tree

12. Try
{
}
System.out.println("hello");
Catch (X e)
{
}

13. Try
{
}
Catch(X e)
{
}
System.out.println("hello");
Catch(Y e)
{
}
CE: catch without try

14. Try
{
}
Catch(X e)
{
}
System.out.println("hello");
Finally
{
}
CE: finally without try

15. Try
{
 Try
 {
 }
 Catch(X e)
 {
 }
 Catch(X e)
 {
 }

16. Try
{
 Try
 {
 }
 Catch(X e)
 {
 }
CE: try without catch or finally

17. Try
{
 Try
 {
 }
 Finally
 {
 }
 Catch(X e)
 {
 }

18. Try
{
}
Catch(X e)
{
 Try
 {
 }
 Finally
 {
 }
}

19. Try
{
}
Catch(X e)
{
 Finally
 {
 }
}
CE: finally without try

20. Try
{
}
Catch(X e)
{
}
Finally
{
 Try
 {
 }
 Catch(X e)
}

21. Try
{
}
Catch(X e)
{
}
Finally
{
 Finally
 {
 }
}
CE: finally without try

22. Try
{
}
Catch(X E)
{
}
Finally
{
}
Finally
{
}
CE: finally with out try

23. Try
 System.out.println("hello");
 Catch (X e)
 {
 System.out.println("catch");
 }
 Finally
 {
 }
→invalid

24. Try
{
}
Catch(X e)
 System.out.println("catch");
 Finally
 {
 }
→invalid

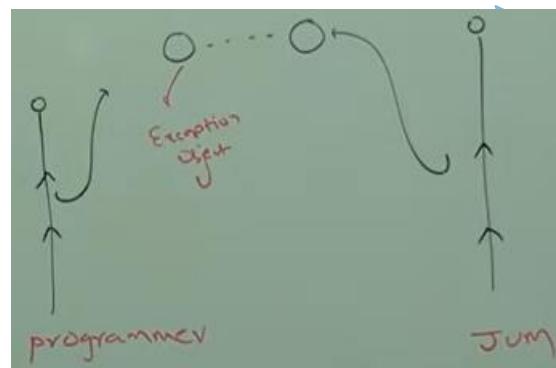
```

25. Try
{
}
Catch(X e)
{
}
Finally
System.out.println("finally");

```

→ invalid

Throw keyword:



Sometimes we can create exception object explicitly we can handover to the JVM manually for this we have to use **throw** keyword

throw new ArthimeticException(" / by Zero")

hand over our created exception object to the JVM manually

creation of arithmeticException object explicitly

Hence the main objective of throw keyword is to handover our created exception object to the JVM manually.

Hence the result of the following two program is exactly same

```

Class Test
{
    Publiclcl static void main(String[ ] args)
    {
        System.out.println(10/0);
    }
}
CE: exception in thread main
java.lang.ArthimeticException : / by zero
at Test main()

```

In this case main method is responsible to create exception object and handover to the JVM

```

Class Test
{
    Publilc staic void main(String[ ] args)
    {
        Throw new
ArithmetricException(" / by zero");
    }
}
CE: exception in thread main
java.lang.ArthimeticException : / by zero
at Test main()

```

In this case programmer creation exception object explicitly and handover to the JVM manually.

NOTE:

Best use of throw keyword is for user defined exception or customized exception

Case 1:

Throw e;
If 'e' refers null then we will get null pointer exception.

```

Class Test
{
    Static ArithmeticException e = new
ArithmetricException();
    Public static void main(String[ ] args)
    {
        Throw e;
    }
}
→RE: ArithmeticException

```

```

Class Test
{
    Static ArithmeticException e ;
    Public static void main(String[ ] args)
    {
        Throw e;
    }
}
→RE: null pointer exception

```

Case 2:

After throw statement we are not allowed to write any statement directly otherwise we will get compile time saying unreachable statement.

```
Class Test
{
    Public static void main(String[ ]
args)
    {
        Systme.out.println(10/0);
        System.out.println("hello");
    }
} →RE: ArithmeticException / by
zero
```

```
Class Test
{
    Public static void main(String[ ]
args)
    {
        Throw new
arithmeticException(" / by zero");
        System.out.println("hello");
    }
} →CE: unreachable statement
```

Case 3:

We can use throw keyword only throwable types if we are trying to use for normal java objects, we will get compile time error saying “incompatible types”

```
Class Test
{
    Public static void main (String []
args)
    {
        Throw new Test ();
    }
} → CE: incompatible types
found: Test
required: java.lang.throwable
```

```
Class Test extends RuntimeException
{
    Public static void main (String[ ]
args)
    {
        Throw new Test ();
    }
} → RE: exception in thread "main" test at
Test main ();
```

Throws keyword:

In our program if there is a possibility of raising checked exception then compulsory, we should handle that checked exception otherwise we will get compile time error saying “unreported exception XXX; must be caught or declared to be thrown”.

Example 1:

```
Import java.io.*;
Class Test
{
    Public static void main(String[ ] args)
    {
        PrintWriter pw = new PrintWriter("abc.txt");
        Pw.println("hello");
    }
}
```

CE: unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown.

Example 2:

```
Class Test
{
    Public static void main(String[ ] args)
    {
        Thread.sleep(10000);
    }
}
```

CE: unreported exception java.lang.Interruptedexception; must be caught or declared to be thrown.

We can handle this compile time error by using the following two ways

1st way:

Try- catch:

```
Class Test
{
    Public static void main (String [ ] args)
    {
        Try
        {
            Thread.sleep(10000);
        }
        catch (InterruptedException e)
        {
        }
    }
}
```

2nd way:

Throws keyword:

We can use throws keyword to delegate responsibility of exception handling to the caller (it may be another method or JVM) then caller method is responsible to handle that exception.

Class Test

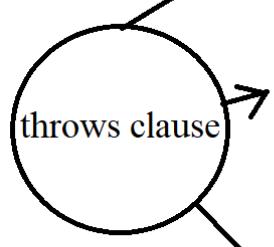
```
{  
    Public static void main (String[ ] args) throws InterruptedException  
    {  
        Thread. Sleep (10000);  
    }  
}
```

Throws keyword required only for checked exceptions and usage of throws keyword for unchecked exceptions there is no use or impact.

Throws keyword required only to convince compiler and usage of throws keyword doesn't prevent abnormal termination of the program.

```
class ThrowsDemo  
{  
    public static void main (String[] args) throws InterruptedException  
    {  
        doStuff ();  
    } CE: unreported exception java.lang.InterruptedException; must be caught or  
declared to be thrown  
    public static void doStuff () throws InterruptedException  
    {  
        doMoreStuff();  
    } CE: unreported exception java.lang.InterruptedException; must be caught or  
declared to be thrown  
    public static void doMoreStuff() throws InterruptedException  
    {  
        Thread.sleep(10000);  
        System.out.println("task completed");  
    } CE: unreported exception java.lang.InterruptedException; must be caught or  
declared to be thrown  
}
```

In the above program if we remove at least one throws statement then the code won't compile

- 
1. we can use to delegate responsibility of exception handling to the caller (it may be method or JVM)
 2. it is required only for checked exception and usage of throws keyword for unchecked exception there is no impact
 3. it is required only to convince compiler and usage of throws does not prevent abnormal termination of program

Note: it is recommended to use try catch over throws keyword

Case 1:

We can use throws keyword for method and constructors but not for class

```
Class Test → invalid
{
    Test() throws Exception → valid
    {
    }
    Public void m1() throws Exception → valid
    {
    }
}
```

Case 2:

We can use throws keyword only for throwable types if we are trying to use for normal java classes then we will get compile time error saying “ incompatible types”

```
Class Test
{
    Public void m1() throws Test
    {
    }
}
→CE: incompatible types ;
Found: Test
Required: java.lang.throwable
```

Class test extends RuntimeException
{
 Public void m1() throws Test
 {
 }
} → valid

Case 3:

```
class Test
{
    public static void main(String[] args)
    {
        throw new Exception(); // checked
    }
}
→CE: unreported exception
java.lang.Exception; must be caught or
declared to be thrown
```

```
class Test369
{
    public static void main(String[] args)
    {
        throw new Error(); // unchecked
    }
}
→RE: exception in thread main
java.lang.Error at Test.main()
```

Case 4: ****

With in the try block there is no chance of raising an exception we can't write catch block for that exception otherwise we will get compile time error saying "Exception XXX is never thrown in body of corresponding try statement". But this rule is applicable for fully checked exceptions.

```
class Test369
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("hello");
        }
        catch (ArithmaticException e) // unchecked
        {
        }
    }
}
```

} → hello

```
class Test369
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("hello");
        }
        catch (Exception e)// partially checked
        {
        }
    }
}
```

} → hello

```

import java.io.*;
class Test369
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("hello");
        }
        catch (IOException e) // fully checked
        {
        }
    }
}

```

→ CE: exception java.IOexception Is never thrown in body of corresponding try statement

```

import java.io. *;
class Test369
{
    public static void main (String [] args)
    {
        try
        {
            System.out.println("hello");
        }
        catch (InterruptedException e) // fully checked
        {
        }
    }
}

```

→ CE: exception java.lang. InterruptedException is never thrown in body of corresponding try statement

```

class Test369
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("hello"); //unchecked
        }
        catch (error e)
        {
        }
    }
}

```

→ hello

Exception handling keywords summary

1. **Try:** to maintain risky code
2. **Catch:** to maintain exception handling code
3. **Finally:** to maintain clean up code
4. **Throw:** to hand-over our created exception object to the JVM manually
5. **Throws:** to delegate responsibility of exception handling to the caller

Various possible compile time errors in exception handling

1. Unreported exception XXX ; must be caught or declared to be thrown
2. Exception XXX has already been caught;
3. Exception XXX is never thrown in body of corresponding try statement
4. unreachable statement
5. incompatible types
Found: Test
Required: java.lang.throwable
6. try without catch or finally
7. catch without try
8. finally without try

customized or user defined exceptions

some time to meet program a requirement we can define our own exception such type of exceptions is called customized or user defined exception.

Example:

TooYoungException, ToOldException, InSufficientFundsException etc.,

```
class TooYoungException extends RuntimeException
{
    TooYoungException(String s)
    {
        super(s);
    }
}

class TooOldException extends RuntimeException
{
    TooOldException (String s)
    {
        super(s); // to make description available to default exception handler
    }
}
```

Customized Exception
or user defined
exception

```

class CustExceptionDemo
{
    public static void main(String[] args)
    {
        int age = Integer.parseInt(args[0]);
        if (age>60)
        {
            throw new TooYoungException("please wait some more time  

definitely you will get best match");
        }
        else if (age<18)
        {
            throw new TooOldException("your age already crossed marriage age and  

no chance getting marriage");
        }
        else
        {
            System.out.println("you will get match details soon by email...!");
        }
    }
}

```

Note:

1. throw key word is best suitable for user defined and customized exception but not for predefined exceptions.
2. It is highly recommended to define customized exceptions as unchecked that is we have to extends runtime exception but not exception

Top 10 exception in java

Based on the person who is raising an exception all exceptions are divided into two categories

1. JVM exceptions
2. Programmatic exceptions

JVM Exceptions:

The exceptions which are raised automatically by JVM whenever particular event occurs are called JVM exceptions

Example:

ArithmaticException, NullPointerException etc.,

Programmatic exceptions:

The exceptions which are raised explicitly either by programmer or API developer to indicate that something goes wrong are called programmatic exceptions

Example:

TooOldException, IllegalArgumentException etc.,

ArrayIndexOutOfBoundsException:

It is the child class runtime exception and hence it is unchecked

Raised automatically by JVM whenever trying to access array element with out of range index

Example:

```
Int[ ] x = new int[ 4];
System.out.println(x[0]); 0
System.out.println(x[10]); RE: ArrayIndexOutOfBoundsException
System.out.println(x[-10]); RE: ArrayIndexOutOfBoundsException
```

NullPointerException:

It is the child class of runtime exception and hence it is unchecked.

Raised automatically by JVM whenever we are trying to perform any operation on null

Example:

```
String s = null;
System.out.println(s.length()); RE: NullPointerException
```

ClassCastException:

It is the child class of runtime exception and hence it is unchecked

Raised automatically by JVM when ever we are trying to typecast parent object to child type

Example:

```
String s = new String("malachi");
Object o = (Object)s; → valid

Object o = new Object();
String s = (String)o → RE: classCastException
```

```
Object o new String("malachi");
String s = (String)o; → valid
```

StackOverflowError:

It is child class of error and hence it is unchecked

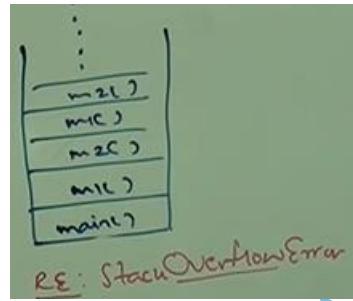
Raised automatically by JVM whenever we are trying to perform recursive method call

Example:

Class Test

```
{  
    Public static void m1()  
    {  
        M2();  
    }  
    Public static void m2()  
    {  
        M1();  
    }  
    Public Static void main(String [ ] args)  
    {  
        M1();  
    }  
}
```

→ RE: StackOverflowError



NoClassDefFoundError:

It is the child class of error and hence it is unchecked

Raised automatically by JVM whenever JVM unable to find required .class file

Example:

Java Test

If Test class file not available then we will get runtime exception saying
NoClassDefFoundError: Test

ExceptionInInitializerError:

It is the child class of Error and hence it is unchecked

Raised automatically by JVM if any exception occurs while executing static variables assignments and static blocks.

Example 1;

Class Test

```
{  
    Static int x = 10/0;  
}  
} RE: ExceptionInInitializerError caused by : java.lang.ArthematicException / by zero
```

```

Class Test
{
    Static
    {
        String s = null;
        System.out.println(S.length());
    }
} → RE: ExceptionInInitializerError caused by : java.lang.NullPointerException

```

IllegalArgumentException:

It is the child class of runtime exception and hence it is unchecked

Raised explicitly either by programmer or by API developer to indicate that a method has been invoked with illegal argument

Example:

The valid range of thread priority is 1 to 10 if we are trying to set the priority with any other value then we will get run time exception saying illegal argument exception

```

Thread t = new Thread();
t.setPriority(7); → valid
t.setPriority(15); → RE: IllegalArgumentException

```

NumberFormatException:

It is the direct child class of illegal argument exception which is the child class of runtime exception and hence it is a unchecked.

Raised explicitly either by programmer or API developer to indicate that we are trying to convert string to number and the string is not properly formatted

Example:

```

Int I = Integer.parseInt("10");
Int I = Integer.parseInt("ten"); → RE: NumberFormatException

```

IllegalStateException:

It is the child class of runtime exception and hence it is unchecked

Raised explicitly either programmer or API developer to indicate that a method has been invoked at wrong time

Example:

After starting of a thread, we are not allowed to restart the same thread once again otherwise we will get runtime exception saying illegal thread state exception.

<pre> Thread t = new Thread(); t.start(); → valid </pre>	... <pre> t.start(); → RE: IllegalThreadStateException </pre>
--	--

AssertionError:

It is the child class of error and hence it is unchecked

Raised explicitly by the programmer or API developer to indicate that Assert statement fails

```
assert (x > 10);
```

If x is not greater than 10 then we will get runtime exception saying “AssertionError”.

Exception / Error	Raised By
1. ArrayIndexOutOfBoundsException	Raised automatically by JVM and hence these are JVM exception
2. NullPointerException	Raised automatically by JVM and hence these are JVM exception
3. ClassCastException	Raised automatically by JVM and hence these are JVM exception
4. StackOverflowError	Raised automatically by JVM and hence these are JVM exception
5. NoClassDefFoundError	Raised automatically by JVM and hence these are JVM exception
6. *** ExceptionInInitializerError	Raised automatically by JVM and hence these are JVM exception
7. IllegalArgumentException	Raised explicitly either by programmer or by API developer and hence these are programmatic Exception
8. NumberFormatException	Raised explicitly either by programmer or by API developer and hence these are programmatic Exception
9. IllegealStateException	Raised explicitly either by programmer or by API developer and hence these are programmatic Exception
10. AssertionError	Raised explicitly either by programmer or by API developer and hence these are programmatic Exception

1.7 version enhancements with respect to exception handling

as the part of 1.7 version in exception handling the following two concepts introduced

1. try with resources
2. multi-catch blocks

try with resources:

until 1.6 version it is highly recommended to write finally block to close resources which are open as the part of try block

```
BufferedReader br = null  
Try  
{  
    Br = new BufferedReader(new FileReader("input.txt"));  
    // use br based on out requirement  
}  
Catch(IOException e)  
{  
    // handling code  
}  
Finally  
{  
    If(br != null)  
    {  
        Br.close();  
    }  
}
```

```
}
```

The problem in this approach is compulsory programmer is required to close inside finally block it increases complexity of programming.

We have to write finally block compulsory and hence it increases length of the code and reduces readability.

To overcome above problems sun people introduced try with resources in 1.7 version.

The main advantage of try with resources is whatever resources is open as the part of try block will be close automatically once control reaches end of try block either normally or abnormally and hence, we are not required to close explicitly so that complexity of programming will be reduced.

We are not required to write finally block so that length of the code will be reduced and readability will be improved.

```
Try(BufferReader br = new BufferedReader(new FileReader("input.txt")))
{
    // user br based on our requirement
    Br will be closed automatically once control reaches end of try block either normally
    or abnormally and we are not responsible to close explicitly.
}
Catch (IOException e)
{
    //handling code
}
```

Conclusions:

1. we can declare multiple resources but these resources should be separated with semicolon(;)

```
try (R1;R2;R3)
{
}
```

Example:

```
Try(FileWriter fw = new FilerWriter("output.txt"); FileReader fr = new
FileReader("input.txt")
{
```

2. all resources should be autocloseable resources.
 - a) A resources said to be auto closeable if and only if corresponding class implements java.lang.AutoCloseable interface.
 - b) All IO related resources, database related resources, network related resources or already implemented auto closeable interface.
 - c) Being a programmer, we are not required to do any thing we should aware the point.

- d) Auto closeable interface came in 1.7 version and it contains only one method close “**public void close()**”;
- 3. All resource reference variables are implicitly final and hence within the try block we cannot perform reassignment otherwise we will get compile time error

Example:

```
Import java.io.*;
Class TryWithResources
{
    Public static void main(String[ ] args) throws Exception
    {
        Try(BufferedReader br = new BufferedReader(new FileReader("input.txt")))
        {
            Br = new BufferedReader(new FileReader("input.txt"))
        }
    }
}→CE: auto closeable resource br may not be assigned
```

- 4. Until 1.6 version try should be associated with either catch or finally but 1.7 version onwards we can take only try with resources without catch or finally

```
Try(R)
{
}
```

Summary:

The main advantage with resources is we are not required to write finally block explicitly we are not required to close explicitly hence until 1.6 version finally block is just like hero but from 1.7 version onwards it is dummy and becomes zero.

Multi-catch block

Until 1.6 version even though multiple different exceptions having same handling code for every exception type we have to write a separate catch block. It increase length of the code and reduces readability.

```
try
{
}

catch (ArithimeticException e)
{
    e.printStackTrace();
}

catch( IOException e)
{
    e.printStackTrace();
}

catch(NullPointerException e)
{
    System.out.println(e.getMessage());
}

catch (InterruptedException e)
{
    System.out.println(e.getMessage());
}
```

To overcome this problem sun people introduce multi catch block in 1.7 version

According to this we can write a single catch block that can handle multiple different type of exceptions

```
Try
{
}

Catch (ArithimeticException | IOException )
{
    e.printStackTrace();
}

Catch(NullPointerException | InterruptedException E)
{
    System.out.println(e.getMessage());
}
```

The main advantage of this approach is length of the code will be reduced and readability will be improved.

Example:

```
Import java.io.*;
Class MultiCatchBlock
{
```

```

Public static void main(String[ ] args)
{
    Try
    {
        System.out.println(10/0);
        String s = null;
        System.out.println(s.length());
    }
    Catch(ArithmeticException | NullPointerException e)
    {
        System.out.println(e);
    }
}

```

In the above example whether raised exception is either Arithmetic Exception or NullPointerException the same catch block can listen

In multi catch block there should not be any relation between exception types(either child to parent or parent to child or same type) otherwise we will get compile time error.

```

Try
{
}

Catch(arithmeticException | Exception e)
{
    e.printStackTrace();
}

```

→ CE: alternatives in a multi catch statement cannot be related by sub classing

Exception propagation

Inside a method if an exception raised and if we are not handling that exception then exception object will be propagated to caller than the caller method is responsible to handle the exception this process is called exception propagation

Rethrowing exception:

We can use this approach to convert one exception type to another exception type.

```

Try
{
    System.out.println(10/0);
}
Catch (ArithmeticException e){
    Throw new NullPointerException();
}

```

Multithreading

1. Introduction
2. The ways to define a thread
 - a. By extending thread class
 - b. By implementing Runnable(I)
3. Getting and setting name of thread
4. ****Thread priorities
5. The methods to prevent thread execution
 - a. Yield ()
 - b. Join ()
 - c. Sleep ()
6. Synchronization
7. Interthread communication
8. Deadlock
9. Daemon threads
10. Multithreading enhancement

Introduction

Multitasking:

Executing several tasks simultaneously is the concept of multitasking. There are two types of multitasking

1. Process based multitasking
2. Thread based multitasking

Process based multitasking:

Executing several tasks simultaneously where each task is separate independent program(process) is called process-based multitasking.

Example:

While type 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.

Processes base multitasking is best suitable at OS level

Thread based multitasking:

Executing several tasks simultaneously where each task is separate independent part of the same program is called thread base multitasking and each independent part is called a thread.

Thread based multitasking is best suitable at programmatic level.

Whether it is process based are thread based the main objective of multi-tasking is to reduce response of the system and improve performance.

The main important application areas of multi-threading are

1. To develop multimedia graphics
2. To develop animations
3. To develop video games
4. To develop web servers and application servers etc.,

When compared with old language developing multi-threaded application in java is very easy. Because java provides in built support for multithreading with rich API (Thread, Runnable, ThreadGroup...).

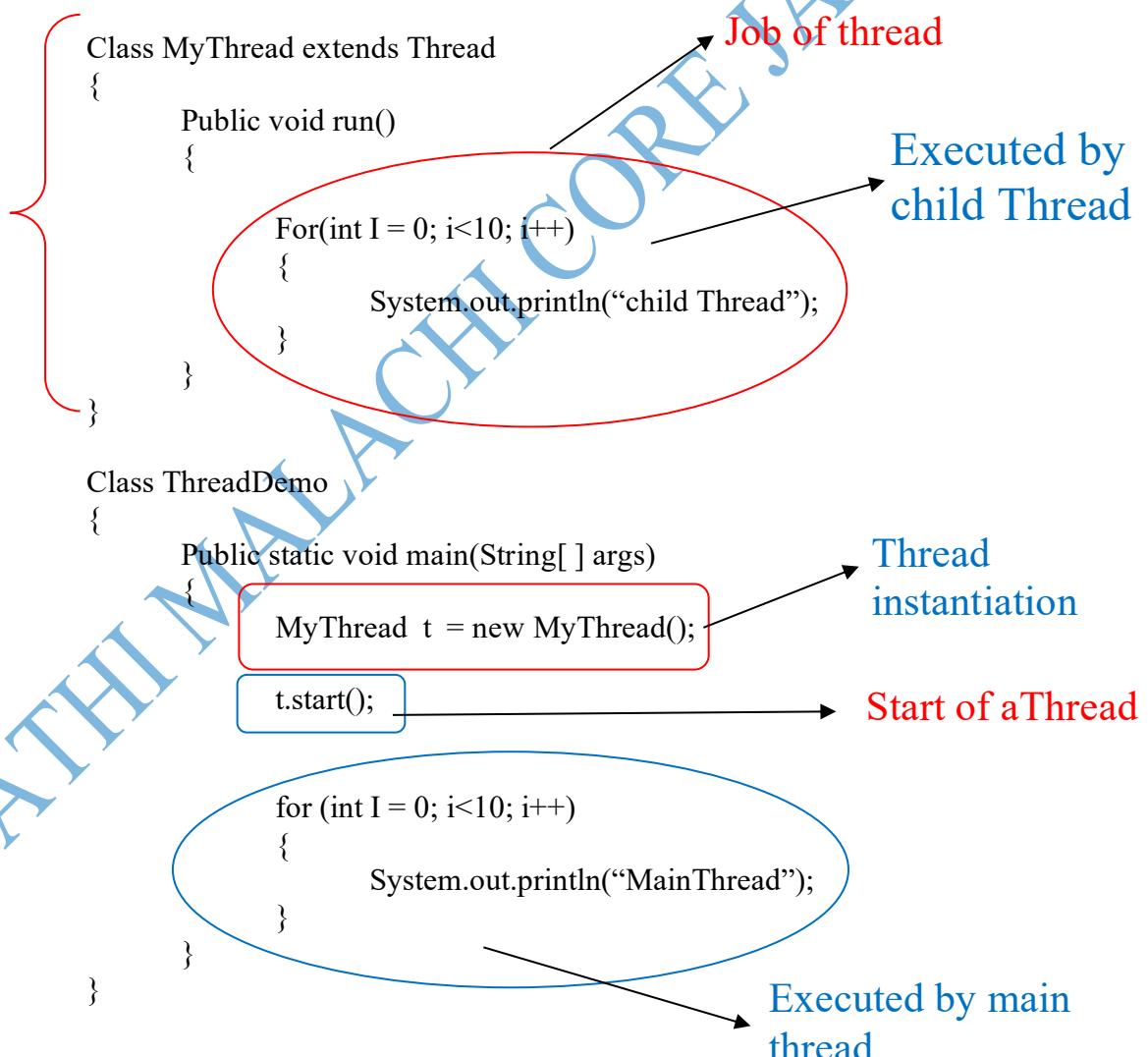
Defining a thread:

We can define a thread in the following two ways

1. By extending thread class
2. By implementing runnable interface

By extending thread class:

Defining
a thread



Case 1: **Thread scheduler**

It is the part of JVM

It is responsible to schedule threads that is if multiple threads are waiting to get chance of execution then in which order threads will be executed is decided by thread scheduler.

We cannot expect exact algorithm followed by thread scheduler it is varied from JVM to JVM hence we cannot expect thread execution order and exact output

Hence whenever situation comes to multi-threading there is no guarantee for exact output but we can provide several possible outputs

The following are various possible outputs for the above program

p-1	P-2	P-3	P-4
main thread	Child thread	main thread	Child thread
main thread	Child thread	Child thread	main thread
main thread	Child thread	main thread	Child thread
main thread	Child thread	Child thread	main thread
.....
.....
.....
Child thread	main thread	main thread	main thread
Child thread	main thread	main thread	Child thread
Child thread	main thread	main thread	main thread
Child thread	main thread	main thread	main thread

Case 2: **Difference between t.start() and t.run()**

In the case t.start() a new thread will be created which is the responsible for the execution of run() method

But in the 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

Child thread	{	This total output produces by only main thread
Child thread		
.....		
.....		
.....		
main thread		
main thread		

This total output produced by only main thread

Case 3: **Importance of thread class start method**

Thread class start method is responsible to register the thread with thread scheduler and all 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 multi-threading

```
Start()
{
    1. Register this thread with thread scheduler
    2. Perform all other mandatory activities
    3. Invoke run()
}
```

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

```
Overloaded methods
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("NO-ARG method");
    }
    public void run(int i)
    {
        System.out.println("Int-ARG method");
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

Output:
NO-ARG method

Case 5: If we are not overriding run() method

If we are not overriding run() method then thread class run() method will be executed which as empty implementation hence we won't get any output

```
class MyThread extends Thread  
{  
  
}  
class ThreadDemo  
{  
    public static void main(String[] args)  
    {  
        MyThread t = new MyThread();  
        t.start();  
    }  
}
```

Output:
No output

Note: it 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 over start method will be executed just like a normal method call an new thread won't be created.

```
class MyThread extends Thread  
{  
    public void start()  
    {  
        System.out.println("start method");  
    }  
    public void run()  
    {  
        System.out.println("run method");  
    }  
}
```

```
class ThreadDemo  
{  
    public static void main(String[] args)  
    {  
        MyThread t = new  
        MyThread();  
        t.start();  
        System.out.println("main  
method");  
    }  
}  
  
Output:  
Start method  
Main method(produced by only main thread)
```

Note: it is not recommended to override start () method otherwise don't go for multithreading concept

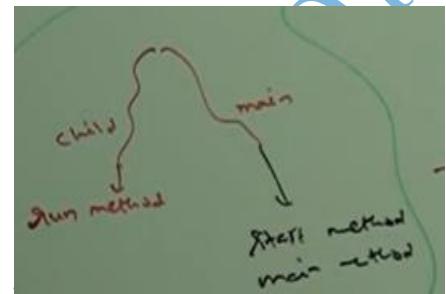
Case 7:

```
class MyThread extends Thread  
{  
    public void start()  
    {  
        super.start();  
        System.out.println("start method");  
    }  
    public void run()  
    {  
        System.out.println("run method");  
    }  
}  
class ThreadDemo  
{  
    public static void main(String[] args)  
    {  
        MyThread t = new MyThread();  
        t.start();  
        System.out.println("main method");  
    }  
}
```

Output:

p-1
run method
start method
main method

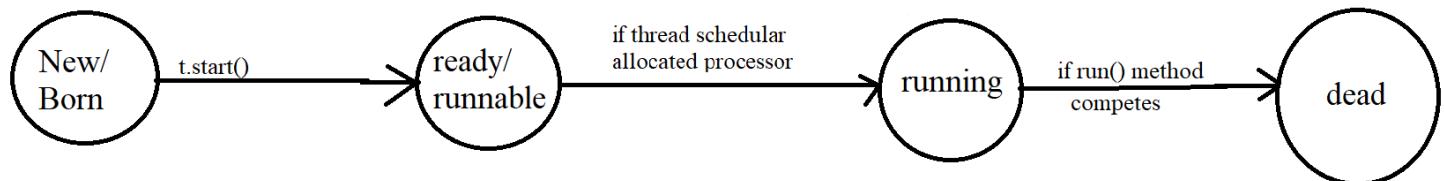
p-2
start method
main method
run method



p-3
start method
run method
main method

Case 8: Thread life cycle

MyThread t = new MyThread()



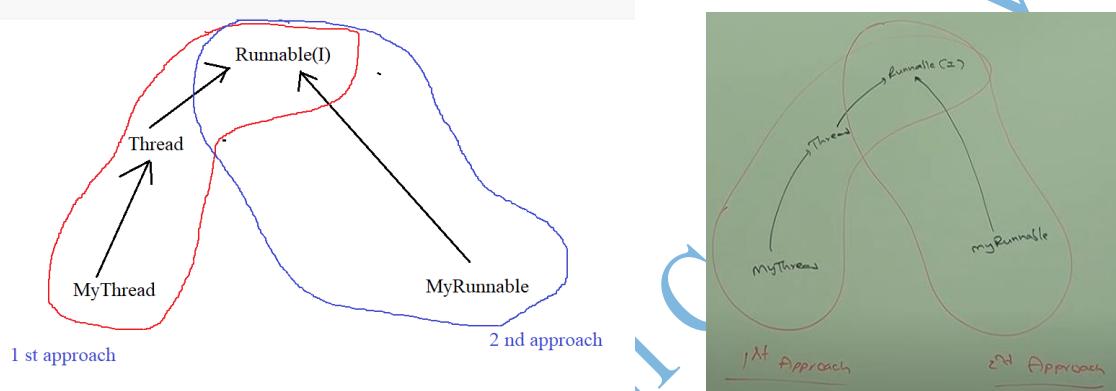
Case 9:

After starting a thread if we are trying to restart the same thread then we will get runtime exception saying "IllegalThreadStateException".

```
Thread t = new Thread();  
t.start();  
.....  
.....  
.....  
t.start(); → RE: IllegalThreadStateException
```

defining a thread by implementing Runnable(I)

we can define a thread by implementing runnable interface



Runnable(I) interface present in java.lang package and it contains only one method that is run() method

Public void run()

defining a thread

```
class MyRunnable implements Runnable  
{  
    public void run()  
    {  
        for (int i=0;i<10 ;i++ )  
        {  
            System.out.println("child thread");  
        }  
    }  
}
```

job of a thread

executed by child thread

```
class ThreadDemoOne  
{  
    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");  
    }  
}
```

target Runnable

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 get compile time error saying MyRunnable class doesn't have start capability
CE: cannot find symbol

Symbol: method start

Location: class MyRunnable

Case 6: r.run();

No new thread will be created and MyRunnable run() method will be executed like normal method call

Which approach is best to define a thread?

Among two ways of defining a thread implements runnable approach is recommended

In the first approach over class 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 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(String name);
4. Thread t = new Thread(Runnable r, String name);
5. Thread t = new Thread(ThreadGroup g, String name);
6. Thread t = new Thread(ThreadGroup g, Runnable r);
7. Thread t = new Thread(ThreadGroup g, Runnable r, String name)
8. Thread t = new Thread(ThreadGroup g, Runnable r, String name, long stacksize)

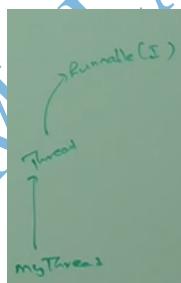
Durga's approach to define a Thread (not recommended to use)

```
Class Mythread extends Thread
{
    Public void run()
    {
        System.out.println("child Thread");
    }
}
Class ThreadDemo
{
    Public static void main(String[] args)
    {
        MyThread t = new MyThread();
        Thread t1 = new Thread(t);
        T1.start();
        System.out.println("main thread")
    }
}
```

Output:

p-1
Child thread
Main thread

p-2
Main thread
Child thread



Getting and 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 and set name of a thread by using the following two methods of thread class

1. Public final String getName()
2. Public final void setName(String name)

```
class MyThread extends Thread  
{  
  
}  
class ThreadDemoTwo  
{  
    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("bhupathi malachi");  
        System.out.println(Thread.currentThread().getName()); → bhupathi malachi  
    }  
}
```

We can get current executing thread object by using Thread.currentThread() method

```
class MyThread extends Thread  
{  
    public void run()  
    {  
        System.out.println(" this run method line executed by thread:"  
+Thread.currentThread().getName());  
    }  
}  
class ThreadDemoThree  
{  
    public static void main(String[] args)  
    {  
        MyThread t = new MyThread();  
        t.start();  
        System.out.println(" this main method line executed by thread:"  
+Thread.currentThread().getName());  
    }  
}
```

Output:

```
this main method line executed by thread: main  
this run method line executed by thread: Thread-0
```

Thread priorities

Every thread in java has some priority it may be default priority generated by JVM
customized priority provided by programmer

The valid range thread priorities is 1 to 10 where 1 is MIN_PRIORITY 10 is MAX_PRIORITY

Thread class define the following constants to represent some standard priorities

Thread.MIN_PRIORITY → 1

Thread.NORM_PRIORITY → 5

Thread.MAX_PRIORITY → 10

Which are valid or invalid priority from the following?

0 → INVALID

1 → VALID

10 → VALID

Thread.LOW_PRIORITY → INVALID

Thread.HIGH_PRIORITY → INVALID

Thread.MIN_PRIORITY → VALID

Thread.NORM_PRIORITY → VALID

Thread scheduler will user priorities while allocating processor

The thread which is having highest priority will get chance first.

If two threads having same priority then we can not except exact execution order it depends on thread scheduler.

Thread class defines the following methods to get and set priority of a thread

1. Public final int getPriority()

2. Public final void setPriority(int p); allowed values range is 1 to 10
other wise runtime exception: IllegalArgumentException

Example:

t.setPriority(7) → valid

t.setPriority(17) → RE: IllegalArgumentException

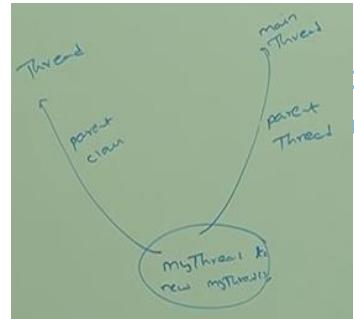
default priority

The default priority only for main thread is 5 but for all remaining threads default priority will be inherited from parent to child. That is whatever priority parent thread has the same priority will be there for child thread.

Example;

```
class MyThread extends Thread  
{  
}  
class ThreadDemoFour
```

```
{  
    public static void main(String[] args)  
    {  
        System.out.println(Thread.currentThread().getPriority());  
        //Thread.currentThread().setPriority(15);  
        Thread.currentThread().setPriority(7);      → line one  
        MyThread t = new MyThread();  
        System.out.println(t.getPriority());  
    }  
}
```



If we comment line one then child thread priority become 5

```
class MyThread extends Thread  
{  
    public void run()  
    {  
        for (int i=0;i<10;i++)  
        {  
            System.out.println("child Thread");  
        }  
    }  
}  
class ThreadDemoFive  
{  
    public static void main(String[] args)  
    {  
        MyThread t = new MyThread();  
        //t.setPriority(10);   → line one  
        t.start();  
        for (int i=0;i<10 ;i++ )  
        {  
            System.out.println("main thread");  
        }  
    }  
}
```

If we are commenting line one then both main and child thread have the same priority 5 and hence we can't expect execution order and exact output

If we are not commenting line one then main thread has a priority 5 and child thread has priority 10 hence child thread will get chance first followed by main thread in this case output is

Child thread

Child thread

.....

.....

.....

.....

Main thread

Main thread

.....

.....

.....

Note:

Some platform won't provide proper support for thread priorities.

We can prevent a thread execution by using the following methods

1. Yield()
2. Join()
3. Sleep()

yield():

yield() method causes to pause current executing thread to give the chance for remaining waiting threads of same priority

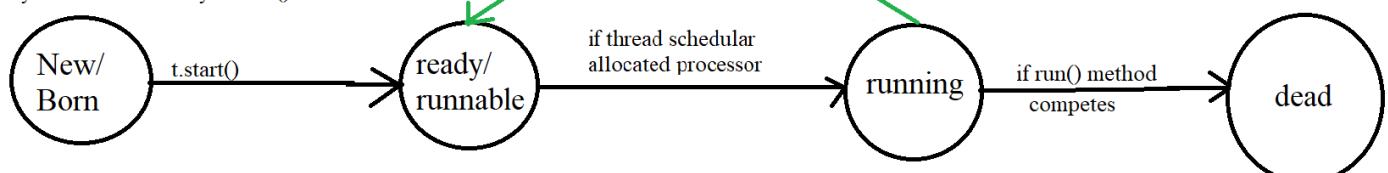
if there is no waiting thread or all waiting threads have low priority then same thread can continue its execution.

If multiple threads are waiting with same priority then which waiting thread will get the chance we can't except it depends on thread scheduler.

The thread which is yielded, when it will get chance once again it depends on thread scheduler and we can't expect exactly

```
public static native void yield();
```

MyThread t = new MyThread()



```

class MyThread extends Thread
{
    public void run()
    {
        for (int i=0;i<10 ;i++ )
        {
            System.out.println("child thread");
            Thread.yield();      →line one
        }
    }
}
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 one then both threads will be executed simultaneously and we can't expect which thread will complete first.

If we are not commenting line one 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.

Some platforms won't provide proper support for yield() method

join() method

if a thread wants to wait until completing some other thread then we should go for join() method.

For example:

If a thread t1 wants to wait until completing t2 than t1 has to call **t2.join()**.

If t1 executes t2.join() then immediately t1 will be entered in to waiting state until t2 completes

Once t2 completes t1 can continue its execution

Example:



Wedding cards printing thread(t2) has to wait until venue fixing thread(t1) completion hence t2 has to call t1.join()

Wedding cards distribution thread(t3) has to wait until wedding cards printing thread(t2) completion hence t3 has to call t2.join()

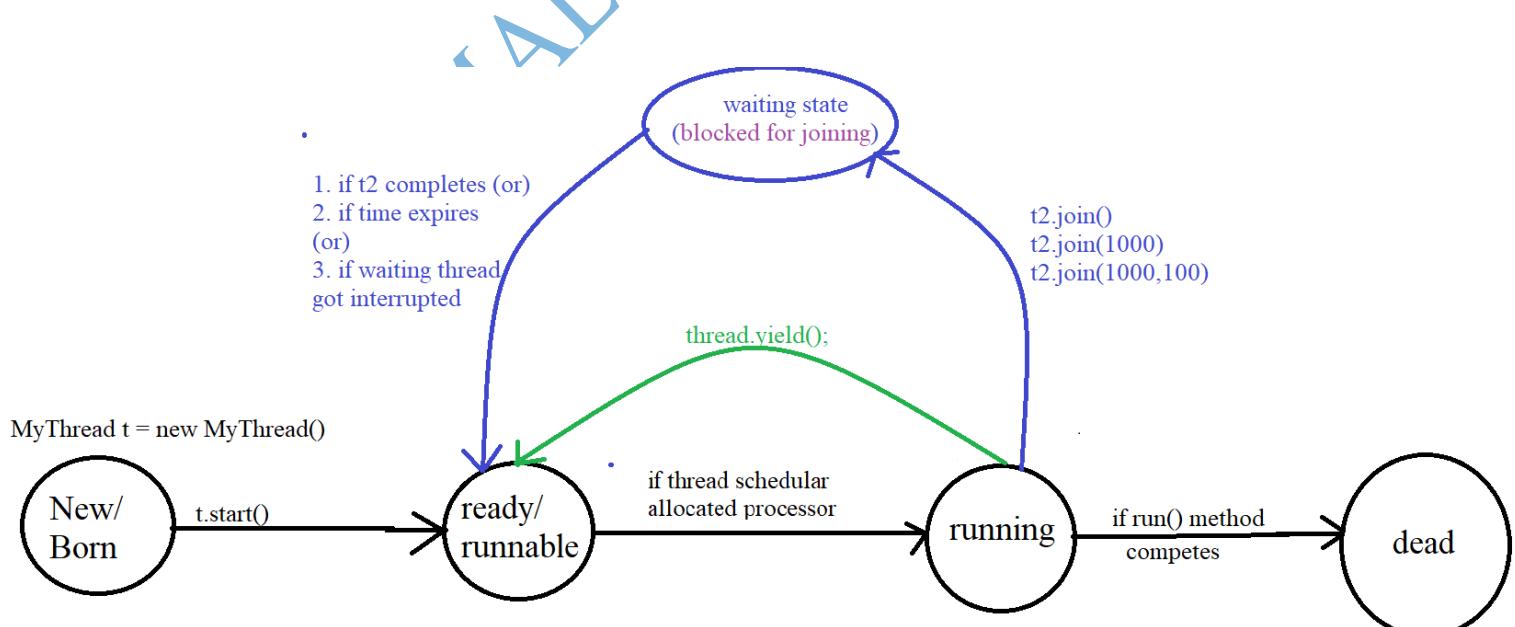
Public final void join() throws InterruptedException

Public final void join(long ms) throws InterruptedException

Public final void join(long ms, int ns) throws InterruptedException

Note:

Every join method throws InterruptedException which is checked exception hence compulsory we should handle this exception either by using try- catch or by throws keywords otherwise we will get compile time error.



Case 1:

Waiting of main thread until completing child thread

EXAMPLE:

```
class MyThread extends Thread
{
    public void run()
    {
        for (int i=0; i<10; i++)
        {
            System.out.println("thread of bhupathi");
            try
            {
                Thread.sleep(2000);
            }
            catch (InterruptedException e)
            {
            }
        }
    }
}
class ThreadJoinDemo
{
    public static void main(String[] args) throws InterruptedException
    {
        MyThread t= new MyThread();
        t.start();
        t.join(); →line one
        for (int i=0; i<10; i++)
        {
            System.out.println("thread of renuka");
        }
    }
}
```

If we commented line one than both main and child thread will be executed simultaneously and we can't expect exact output

If we are not commenting line one the main thread calls join() method on child thread object hence main thread will wait until completing child thread in this case output is

Thread of bhupathi
Thread of bhupathi
Thread of bhupathi
.....
.....

Thread of renuka
Thread of renuka
Thread of renuka

Case 2:

Waiting of child thread until completing main thread

```
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 ThreadJoinDemoTwo
{
    public static void main(String[] args) throws InterruptedException
    {
        MyThread.mt = Thread.currentThread();
        MyThread t = new MyThread();
        t.start();
        for (int i=0;i<10 ;i++)
        {
            System.out.println("main thread");
            Thread.sleep(2000);
        }
        //System.out.println("Hello World!");
    }
}
```

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:

```
main thread
main thread
main thread
.....
.....
Child Thread
Child Thread
Child Thread
```

Case 3:

If main thread call join () method on child thread object and child thread calls join() method on main thread object then both thread will wait for ever and the program will paused or struck up (this is something like dead locked).

Case 4

If a thread call join() method on the same thread it self then the program will be strucked (this is something like dead locked). In this case thread has to wait infinite amount of time

```
class Case4ThreadJoin
{
    public static void main(String[] args) throws InterruptedException
    {
        Thread.currentThread().join();
    }
}
```

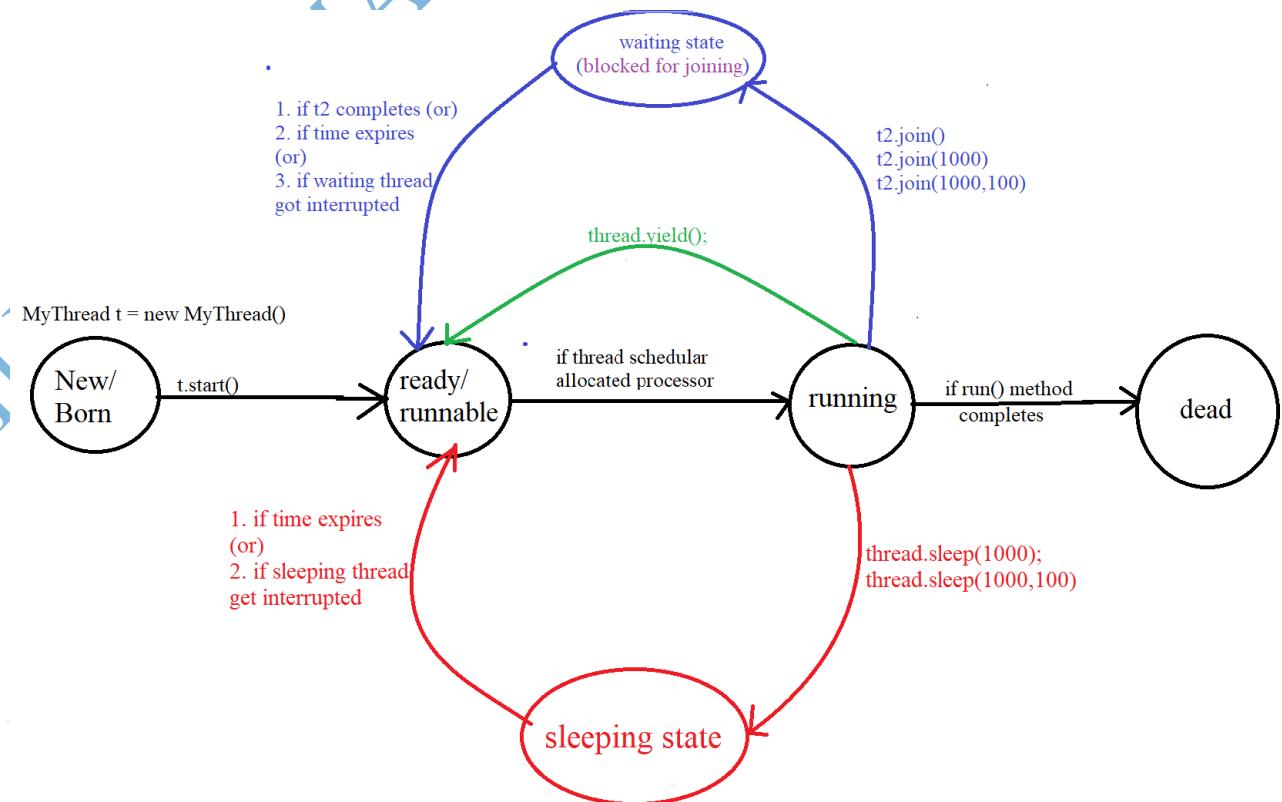
sleep()

if a thread don't want do any operation for a particular amount of time then we should go for sleep() method.

```
public static native void sleep(long ms) throws InterruptedException;
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 compulsory we should handle InterruptedException either by try – catch or by throws keyword other wise we will get compile time error



```

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);
        }
    }
}

```

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()

example:

```

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(); → // line one
        System.out.println("End of Main thread");
    }
}

```

If we comment line one than main thread won't interrupt child thread. In this case child thread execute for loop 10 times.

If we are not commenting line one than main thread interrupts child thread in this case out put is

End of main thread
I am lazy thread
I got interrupted

Note: ****

When ever we are interrupt() method if the target thread not in sleeping state or waiting state then there Is no impact of interrupt call immediately interrupt call will be waited until target thread entered in to sleeping or waiting state

If the target thread entered in to sleeping or wating state then immediately interrupt call we interrupt the target thread

If the target thread never entered into sleeping or waiting state in its life time than there is no impact of interrupt call this is the only case where interrupt call will be wasted.

```
class MyThread extends Thread
{
    public void run()
    {
        for (int i=0;i<100 ;i++)
        {
            System.out.println("i am lazy Thread"+i);
        }
        System.out.println(" i want to sleep");
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {
            System.out.println(" i got interrupted");
        }
    }
}
class ThreadInterruptDemotwo
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();

        //t.interrupt(); // line one
        System.out.println("End of Main thread");
    }
}
```

In the above example interrupt call waited until child thread completes for loop 10000 times.

Comparison table of yield(), join() and 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 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	Yes	No	Sleep(long ms) → native Sleep(long ms, int ns) → non-native
6. Is it static	Yes	No	no

Synchronization

Synchronized is the modifier applicable only for methods and blocks but not for classes and variables.

If multiple threads are trying to operate simultaneously on the same java object than 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 than 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 it increases waiting time of threads and creates performance problems hence if there is no specific requirement than 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 than only lock concept will come into the picture.

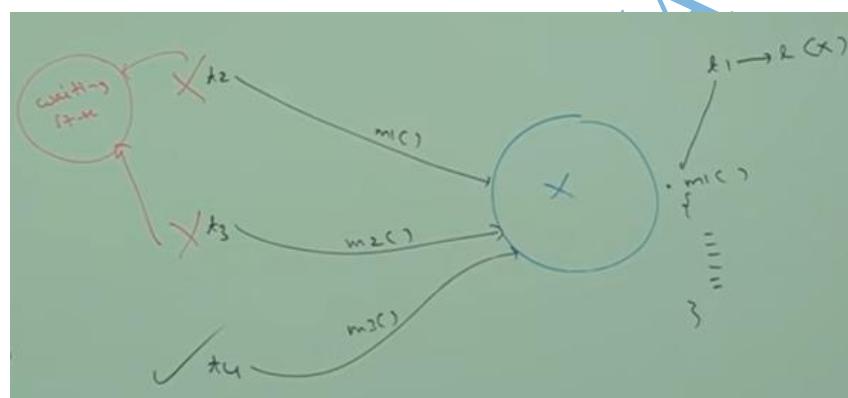
If a thread wants to execute synchronized method on the given object first it has to get lock of that object once thread got the lock than it is allowed to execute any synchronized method on that object

Once method execution completes automatically thread releases the lock.

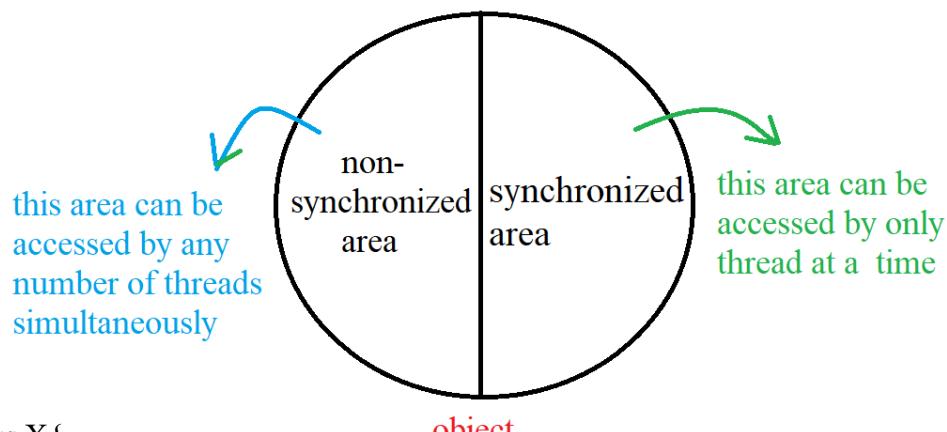
Acquiring and releasing lock internally takes care by JVM and programmer not responsible for this activity.

While a thread executing synchronized method on the given object the remaining threads are not allowed to executes any synchronised method simultaneously on the same object but remaining threads are allowed to execute non- synchronized methods simultaneously

```
Class X
{
    Synch m1();
    Synch m2();
    M3();
}
```



lock concept is implemented based on object but not based on method.



```
Class X{
    Synchronized Are{
        Where ever we are performing update operation (add/ remove/ delete/ replace)
        That is where state of object changing.
    }
    Non-Synchronized area{
        Where ever object state won't be changed like read() operation
    }
}
```

```
class ReservationSystem{  
    Non-synchronized checkAvailability(){  
        Just read operation  
    }  
    Synchronized bookTicket(){  
        Update;  
    }  
}
```

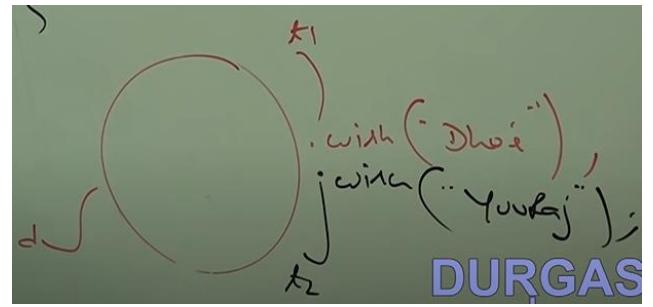
Example:

```
class Display  
{  
    public synchronized void wish(String name)  
    {  
        for (int i=0;i<10 ;i++ )  
        {  
            System.out.print("Good Morning: ");  
            try  
            {  
                Thread.sleep(2000);  
            }  
            catch (InterruptedException e)  
            {  
            }  
            System.out.println(name);  
        }  
    }  
}  
class MyThread extends Thread  
{  
    Display d;  
    String name;  
    MyThread(Display d, String name)  
    {  
        this.d =d;  
        this.name = name;  
    }  
    public void run()  
    {  
        d.wish(name);  
    }  
}
```

```

class SynchronizedDemo
{
    public static void main(String[] args)
    {
        Display d = new Display();
        MyThread t = new MyThread(d, "Dhoni");
        MyThread t1 = new MyThread(d, "yuvraj");
        t.start();
        t1.start();
    }
}

```



If we are not declaring wish() method as synchronized then both threads will be executed simultaneously and hence we will get irregular output

```

Good Morning: Good Morning:yuvraj
Good Morning: dhoni
Good Morning: dhoni

```

If we declare wish() method as synchronized then at a time only one thread is allowed execute wish() method and given display object hence we will get regular output.

```

Good morning : dhoni
Good morning : dhoni
Good morning : yuvraj
Good morning : yuvraj

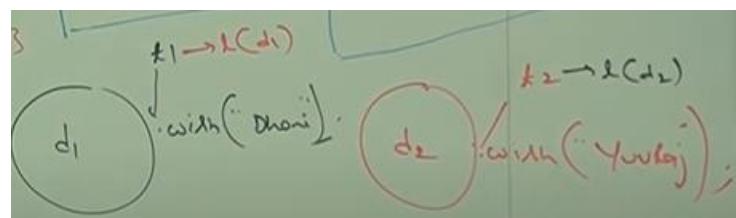
```

Case study:

```

Display d = new Display();
Display d1 = new Display();
MyThread t = new MyThread(d, "Dhoni");
MyThread t1 = new MyThread(d1, "yuvraj");
t.start();
t1.start();

```



even though wish() is synchronize we will get irregular output because thread are operating on different java objects.

Conclusion:

if multiple threads are operating on same java object than synchronization is required

if multiple threads are operating on multiple java objects than synchronization is not required

class level lock

every class in java has a unique lock which is nothing but class level lock.

If a thread wants to execute a static synchronized method than thread required class level lock

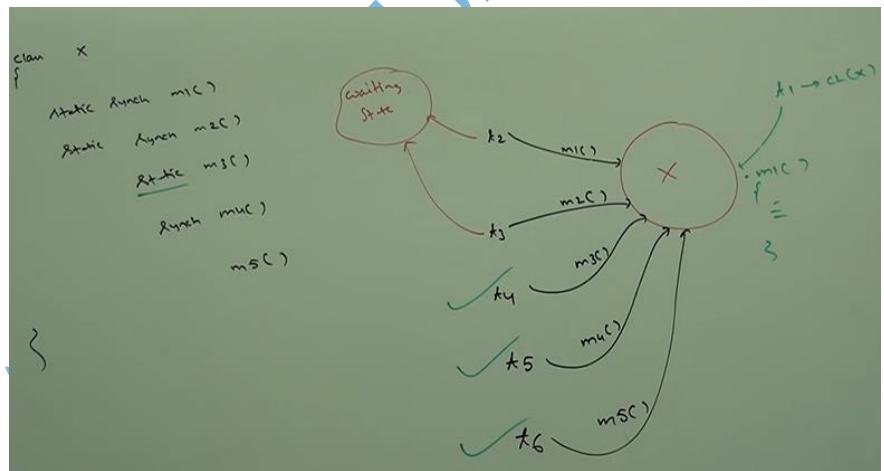
Once thread got class level lock than it is allowed to execute any static synchronized method of that class

Once method execution is complete automatically thread releases the lock.

While a thread executing static synchronized method the remaining threads are not allowed any static synchronized method of that class simultaneously but remaining threads are allowed to execute the following method simultaneously

1. Normal static methods
2. Synchronized instance methods
3. Normal instance methods

```
Class X
{
    Static synchronize m1();
    Static synchronize m2();
    Static m3();
    Synchronize m4();
    M5();
}
```



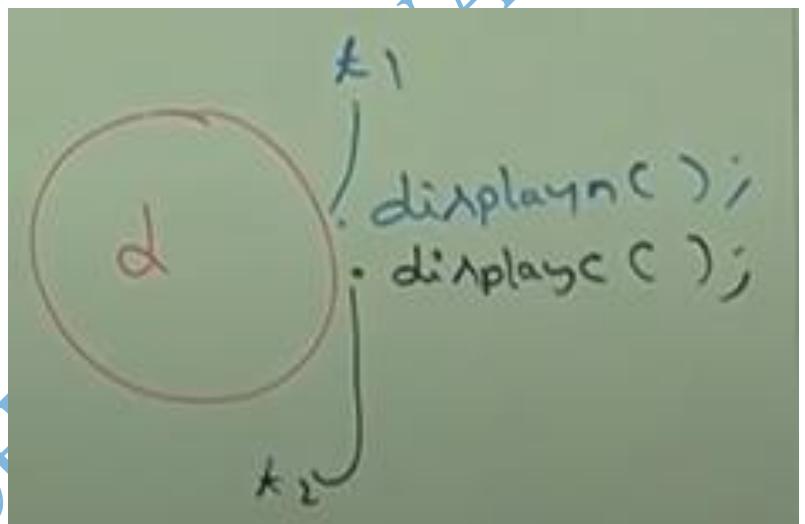
Example 2:

```
class Display
{
    public synchronized void displayNumber()
    {
        for (int i=1; i<10 ;i++)
        {
            System.out.print(i);
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e){}
        }
    }
}
```

```

public synchronized void displayCharacter(){
    for (int i = 65; i<=75 ;i++){
        System.out.println((char)i);
    }
    try
    {
        Thread.sleep(1000);
    }
    catch (InterruptedException e){}
}
class MyThreadOne extends Thread
{
    Display d;
    MyThreadOne(Display d)
    {
        this.d=d;
    }
    public void run()
    {
        d.displayNumber();
    }
}
class MyThreadTwo extends Thread
{
    Display d;
    MyThreadTwo(Display d)
    {
        this.d=d;
    }
    public void run()
    {
        d.displayCharacter();
    }
}
class SynchronizedDemoThree
{
    public static void main(String[] args)
    {
        Display d = new Display();
        MyThreadOne t1 = new MyThreadOne(d);
        MyThreadTwo t2 = new MyThreadTwo(d);
        t1.start();
        t2.start();
    }
}

```



Synchronized block

If very few lines of the code required synchronization than 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 synchronized block over synchronized method is it reduces waiting time of the threads and improves performance of the system.

We can declare synchronized block as follows

1. To Get lock of current object

```
Synchronized(this)
{
    // if a thread got lock of current object then only it is allowed to execute this area.
}
```

2. To get lock of particular object ‘b’.

```
Synchronized(b)
{
    // if a thread got lock of 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 of “Display” class, then only it is allowed to execute
this area
}
```

Example:

```
class Display
{
    public void wish(String name)
    {
        // 1 lakh lines of code assume
        synchronized(this)
        {
            for (int i=0; i<10; i++)
            {
                System.out.print("good morning: " );
                try
                {
                    Thread.sleep(2000);
                }
            }
        }
    }
}
```

```

        catch (InterruptedException e)
        {
        }
        System.out.println(name);
    }
}
//1 lakh line of code assume
}

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 SynchronizedBlockDemo
{
    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 and class types but not for primitives hence we can't pass primitive type as argument to synchronized block otherwise we will get compile time error saying " unexpected type; found: int; required: reference"

```

Int x = 10;
Synchronized(x)
{
}
CE: unexpected type
  Found: int
  Required: reference

```

Frequently asked questions (FAQ)

1. What is synchronized keyword and where we can apply?
2. Explain advantage of synchronized keyword?
3. Explain disadvantage of synchronized keyword?
4. What is race condition?
Ans: 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 and when it is required?
6. What is class level lock and when it is required?
7. What is the difference between class level lock and object level lock?
8. While a thread executing synchronized method on the given object is remaining thread are allowed to execute any other synchronized method simultaneously on the same object?
Ans: no
9. What is synchronized block?
10. How to declare synchronized block to get lock of current object?
11. How to declare synchronized block to get class level lock?
12. What is the advantage of synchronized block over synchronized method?
13. Is a thread can acquire multiple locks simultaneously? → yes. Of course, from different objects

```
Class X
{
    Public synchronized void m1()
    {
        Y y = new Y();
        Synchornized(y)
        {
            → here thread has locks of 'x' and 'y'
            Z z = new Z();
            Synchronizd(z)
            {
                → Here thread has locks of x, y, z
            }
        }
    }
}

X x = new X();
x.m1();
```

14. What is synchronized statement(interview created terminology)?
Ans: the statements present in synchronized method and synchronized block are called synchronized statements.

Inter thread communication

1. Two threads can communicate with each other by using wait(), notify() and notifyAll() methods. The thread which is expecting updation 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, it is responsible to call notify() method then waiting thread will get that notification and continue its execution with those updated items.
2. Wait(), notify(), notifyAll() method present in object class but not in thread class because thread can call these methods on any java object.
3. To call wait(), notify() and notifyAll() methods on any object, thread should be owner of that object that is the thread should have lock of that object that is the thread should be inside synchronized area

Hence, we can call wait(), notify() and notifyAll() methods only from synchronized area otherwise we will get runtime exception saying “IllegalMonitorStateException”.

4. If a thread calls wait() on any object it immediately releases the lock of the particular object and enters into waiting state.
5. If a thread calls notify() method on any object it releases the lock of that object may not immediately except wait(), notify() and notifyAll() there is no other method where thread releases the lock

Method	Is thread releases lock?
Yield()	No
Join()	No
Sleep()	No
Wait()	Yes
Notify()	Yes
notifyAll()	yes

Which of the following is valid?

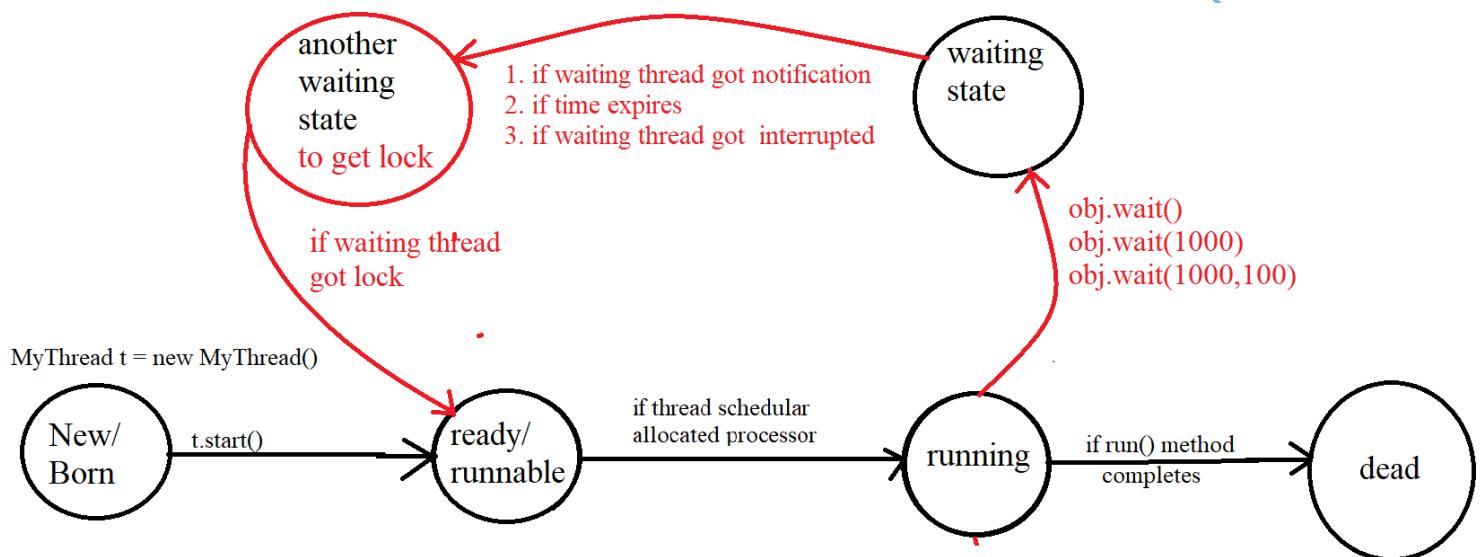
1. If thread calls wait() method immediately it will enter into waiting state without releasing any lock? → invalid
2. If thread calls wait() method it releases the lock of that object but may not immediately? → invalid
3. If thread calls wait() method on any object it releases all locks acquired by the thread and immediately enters into waiting state? → invalid
4. If a thread calls wait() method of any object it immediately releases the lock of the particular object and enters into waiting state? → valid
5. If a thread calls notify() method on any object it immediately releases the lock of the particular object? → invalid
6. If a thread calls notify() method on any object it releases the lock of that object but may not immediately? → valid

Public final void wait() throws InterruptedException
Public final native void wait(long ms) throws InterruptedException
Public final void wait(long ms, int ns) throws InterruptedException

Public final native void notify()
Public final native void notifyAll()

Note:

Every wait() method throws InterruptedException which is checked Exception. Hence whenever we are using wait method compulsory we should handle this InterruptedException either by try-catch or by throws keyword otherwise we will get compile time error.



Example:

```
class InterThreadOne
{
    public static void main(String[] args) throws InterruptedException
    {
        ThreadB b = new ThreadB();
        b.start();
        synchronized(b)
        {
            1. System.out.println("main thread trying to call wait() method");
            b.wait();
            4. System.out.println("main thread got notification");
            5. System.out.println(b.total);
        }
    }
}
```

```

class ThreadB extends Thread
{
    int total = 0;
    public void run()
    {
        synchronized(this)
        {
            2. System.out.println("child thread starts calculation");
            for (int i = 1;i<=100 ;i++ )
            {
                total = total + i;
            }
            3. System.out.println("child thread giving notification");
            this.notify();
        }
    }
}

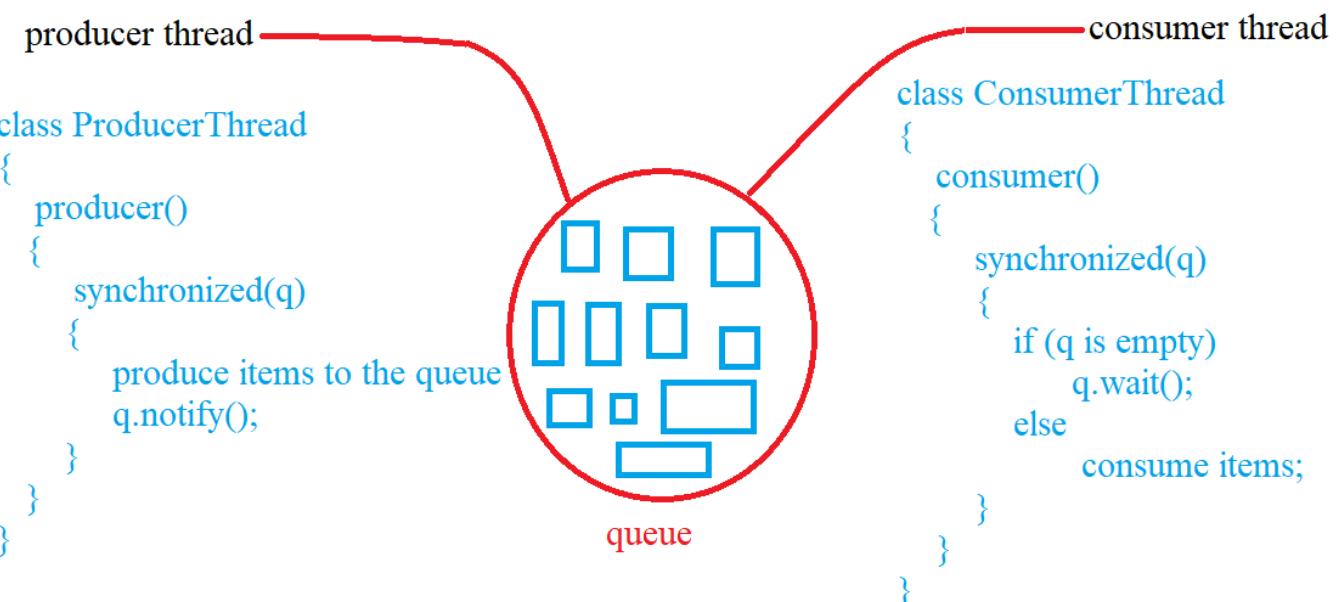
```

Output:

main thread trying to call wait() method
 child thread starts calculation
 child thread giving notification
 main thread got notification
 5050

Producer – consumer problem:

Producer thread is responsible to produce items to the queue and 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.



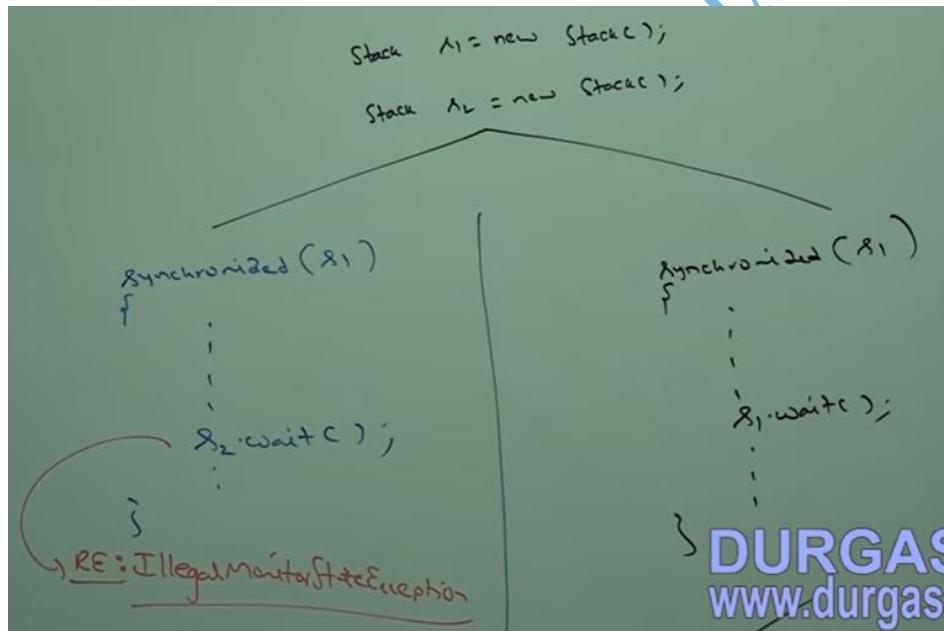
Difference between notify() and notifyAll()

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 have to wait for further notifications. Which thread will be notified we can't except it depends on JVM

We can use notifyAll() to give the notification for all waiting threads of a particular object. Even though multiple threads are notified but execution will be performed one by one because each thread required lock and only one lock is available.

Note:

On which object we are calling wait method thread required the lock of that particular object for example: if we are calling wait() method on s1 then we have to get lock of s1 object but not s2 object



Dead lock

If two thread are waiting for each other forever such type of infinite waiting is called dead lock.

Synchronized keyword is the only reason for dead lock situation hence while using synchronized keyword we have to take special care

There are no resolution techniques for dead lock but several prevention techniques are available.

Example:

```
class A
{
    public synchronized void d1(B b)
    {
        System.out.println("thread1 starts execution of d1() method");
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {
        }
        System.out.println("Thread1 trying to call B's last() method");
        b.last();
    }
    public synchronized void last()
    {
        System.out.println("inside A, last()method");
    }
}
class B
{
    public synchronized void d2(A a)
    {
        System.out.println("thread2 starts execution of d2() method");
        try
        {
            Thread.sleep(5000);
        }
        catch (InterruptedException e)
        {
        }
        System.out.println("Thread2 trying to call A's last() method");
        a.last();
    }
}
```

```

public synchronized void last()
{
    System.out.println("inside B, last() method");
}
}

class DeadLockDemo extends Thread
{
    A a = new A();
    B b = new B();
    public void m1()
    {
        this.start();
        a.d1(b);          // will execute by main thread
    }
    public void run()
    {
        b.d2(a);          // will execute by child thread
    }
    public static void main(String[] args)
    {
        DeadLockDemo t = new DeadLockDemo();
        t.m1();
    }
}

```

Output:

```

thread1 starts execution of d1() method
thread2 starts execution of d2() method
Thread1 trying to call B's last() method
Thread2 trying to call A's last() method

```

In the above program we remove atleast one synchronized keyword then the program entered into dead lock hence synchronized keyword is the only reason for dead lock situation due to this while using synchronized keyword we have to take special care.

Deadlock v/s starvation

Long waiting of a thread where waiting never ends is called deadlock.

Where as long waiting of a thread where waiting ends at certain point is called starvation.

For example: low priority thread has to wait until completing all high priority it may long waiting but ends at certain point, which is nothing but starvation

Daemon threads

The thread which are executing in the background are called daemon threads.

Example:

Garbage collector
Signal dispatcher
Attach listener

The main objective of daemon threads is to provide support for non- daemon threads (main thread). For example: if main thread runs with low memory, then JVM run garbage collector to destroy useless object so that number of bytes of free memory will be improved with this free memory main thread can continue its execution.

Usually, 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()` of `Thread` class.

public Boolean isDaemon()

we can change daemon nature of a thread by using `setDaemon()` method .

public void setDaemon(boolean b)

but changing nature is possible before starting of a thread only. After starting a thread if we are trying to change daemon nature, we will get runtime exception saying “IllegalThreadException”

default nature of thread:

by default, main thread is always non- daemon and for all reaming threads daemon nature will be inherited from parent to child that is if the parent thread is daemon, then automatically child thread is daemon. And if the parent thread is non-daemon, then automatically child thread is also non-demon.

Note:

It is impossible to change daemon nature of main thread because it is already started by JVM at beginning

```
class MyThread extends Thread{ }
class DaemonDemo
{
    public static 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 thread terminates automatically all daemon threads will be terminated irrespective of there position.

```
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 DaemonThreadDemotwo
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.setDaemon(true); //--> line one
        t.start();
        System.out.println("end of main thread");
    }
}
```

If we are commenting line one both main and child are non- daemon and hence both threads will be executed until there completion

If we are not commenting line one then main thread is non-demon and child thread demon hence whenever main thread terminates automatically child thread will be terminated in this case output is

End of main thread
Child thread

End of main thread

Child thread
End of main thread

Java multithreading concept is implemented by using the following two models

1. Green thread model
2. Native OS model

Green thread mode:

The thread which is managed completely by JVM without taking underlying OS support is called green thread. Very few operating systems like Sun Solaris provide support for green thread model. Any way green thread model is deprecated and not recommended to use.

Native OS model:

The thread which is managed by the JVM with the help of underlying OS, is called native OS model. All Windows 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 enters into dead state. Any way `stop()` method is deprecated and not recommended to use.

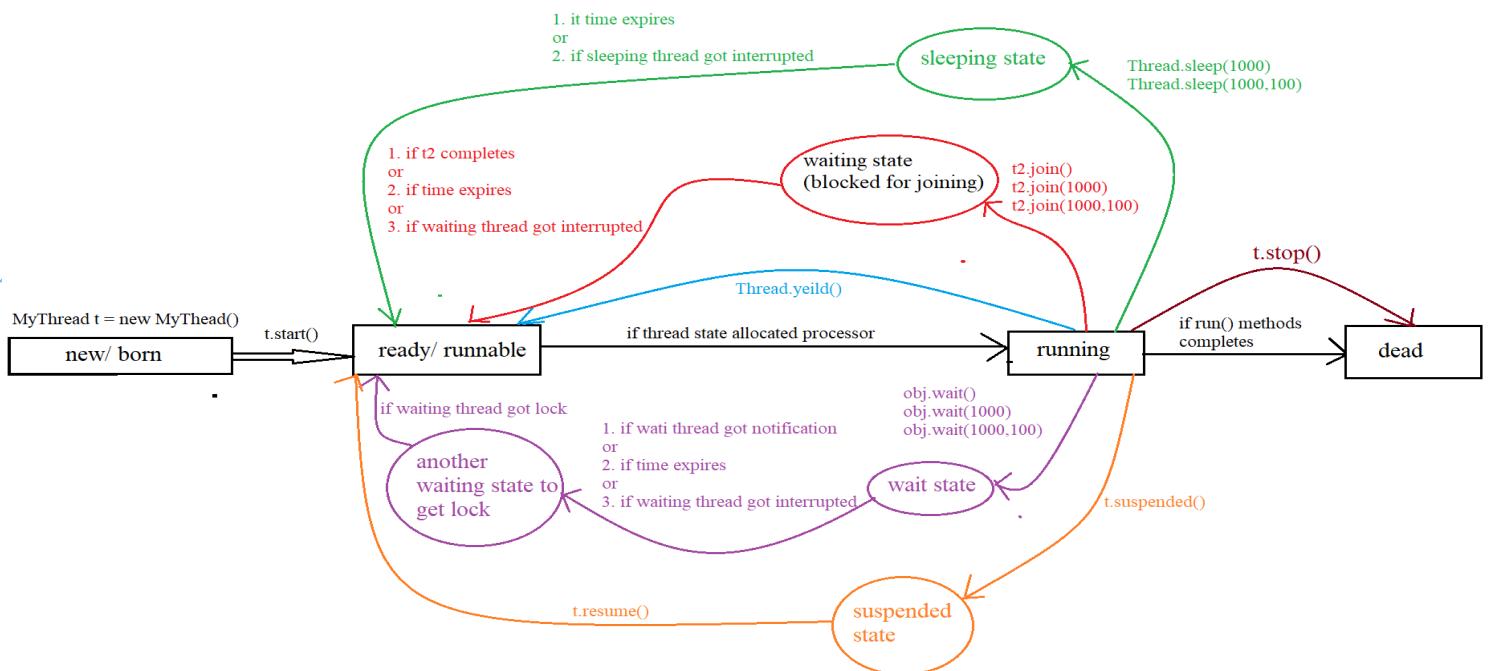
How to suspend and resume of a thread:

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.

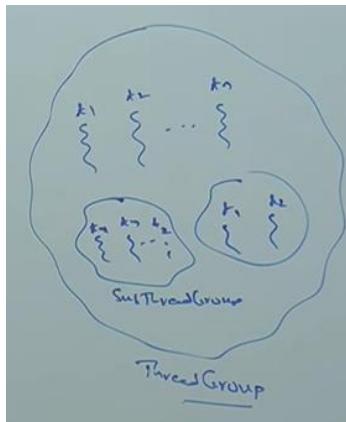


Multithreading enhancement

Thread Group

Based on functionality we can group threads in to a single unit which is nothing but thread group. That is thread group contains a group of threads.

In addition to threads thread group can also contain sub thread groups

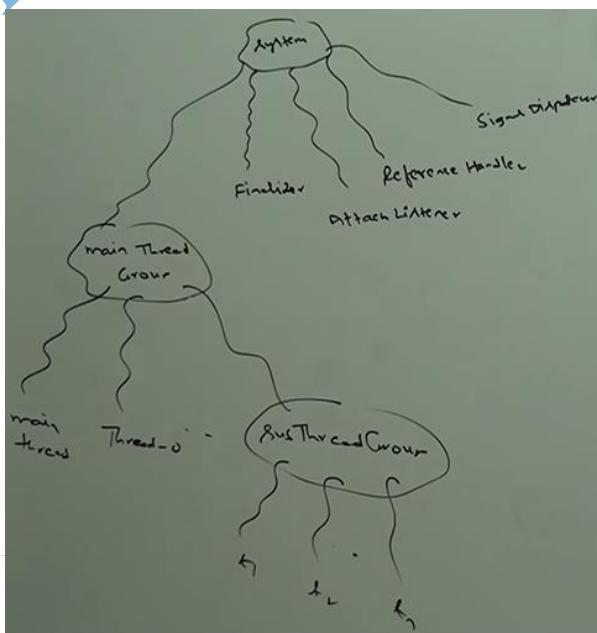


The main advantage of maintaining threads in the form thread group is we can perform common operations very easily.

Every thread in java belongs to some group. Main thread belongs to main group. Every thread group 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

1. Finalizer
2. Reference Handler
3. Signal Dispatcher
4. Attach Listener



```

class ThreadGroupDemo
{
    public static void main(String[] args)
    {
        System.out.println(Thread.currentThread().getThreadGroup().getName());
    //main

    System.out.println(Thread.currentThread().getThreadGroup().getParent().getName());
//system
    }
}

```

Thread Group 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 gname);`

Creates a new thread group with specified group name. the parent of this new group is the thread group of currently executing thread.

Eg: `ThreadGroup g = new ThreadGroup("first group");`

2. `ThreadGroup g = new ThreadGroup(ThreadGroup pg, String GroupName)`

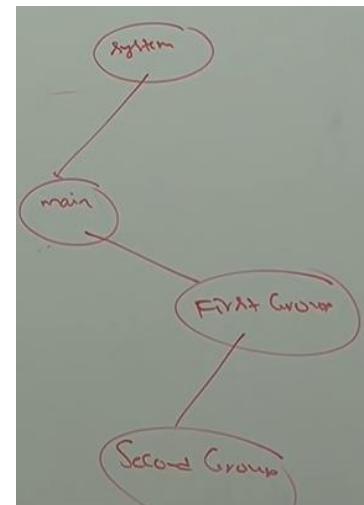
Creates a new `ThreadGroup` with specified group name. the parent of this new thread group is specified parent group.

Eg: `ThreadGroup g1 = new ThreadGroup(g, "second group");`

```

class ThreadGroupDemoOne
{
    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()); //first group
    }
}

```



Important methods of ThreadGroup Class

1. **String getName()**: returns name of the Thread Group
2. **int getMaxPriority()**: returns max priority of thread group
3. **void setMaxPriority(int p)**: to set maximum priority of thread group, the default max priority is 10.

Threads in the thread group that all ready have higher priority won't be affected but for newly added threads this max priority is applicable

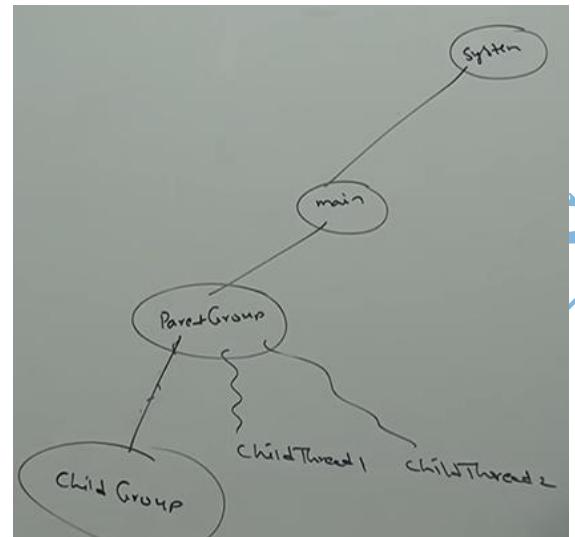
```
class ThreadGroupDemoTwo
{
    public static void main(String[] args)
    {
        ThreadGroup g1= new ThreadGroup("first group");
        Thread t1 = new Thread(g1,"Thread1");
        Thread t2 = new Thread(g1,"Thread2");
        g1.setMaxPriority(3);
        Thread t3 = new Thread(g1,"Thread3");
        System.out.println(t1.getPriority()); //5
        System.out.println(t2.getPriority()); //5
        System.out.println(t3.getPriority()); //3
    }
}
```

4. **ThreadGroup getParent()** : returns parent group of current thread
5. **void list()**: it prints information about thread group to the console.
6. **int activeCount()**: returns number of active thread present in the thread group.
7. **int activeGroupCount()**: it return number of active group present in the current thread group.
8. **int enumerate(Thread[] t)**: to copy all active threads of this thread group into provided thread array. In this case sub thread groups thread will be considered.
9. **Int enumerate(ThreadGroup[] g)**: to copy all active sub thread groups into thread group array.
10. **Boolean isDaemon()**: to check whether the thread group is daemon or not
11. **Void setDaemon(Boolean b)**:
12. **Void interrupt()**: to interrupt all waiting or sleeping threads present in the thread group.
13. **Void destroy()**: to destroy thread group and its sub thread groups. A s

```

class MyThread extends Thread
{
    MyThread(ThreadGroup g, String name)
    {
        super(g, name);
    }
    public void run()
    {
        System.out.println("child thread");
        try
        {
            Thread.sleep(5000);
        }
        catch (InterruptedException e)
        {
        }
    }
}
class ThreadGropuDemoFour
{
    public static void main(String[] args) throws Exception
    {
        ThreadGroup pg = new ThreadGroup("Parent group");
        ThreadGroup cg = new ThreadGroup(pg, "ChildGroup");
        MyThread t1 = new MyThread(pg, "childThread1");
        MyThread t2 = new MyThread(pg, "childThread2");
        t1.start();
        t2.start();
        System.out.println(pg.activeCount());
        System.out.println(pg.activeGroupCount());
        pg.list();
        Thread.sleep(10000);
        System.out.println(pg.activeCount());
        System.out.println(pg.activeGroupCount());
        pg.list();
    }
}

```



Write a program to display all active thread names belongs to system groups and its child groups.

```
class ThreadGroupDemoSix
{
    public static void main(String[] args)
    {
        ThreadGroup system = Thread.currentThread().getThreadGroup().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

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 dead lock.
3. If a thread release a lock then which waiting thread will get that lock we are not having any control on this.
4. There is no API to list out all waiting threads for a lock.
5. The synchronized keyword compulsory we have to use either at method level or with in the method and it is not possible to use across multiple methods.

To over come these problems sun people introduced java.util.concurrent.locks package in 1.5 version.

It also provides several enhancements to the programmer to provide more control on concurrency

Lock interface:

Lock object is similar to implicitly lock acquired by thread to execute synchronized method or synchronized block

Lock implementation provide more extensive operation than traditional implicit locks

Important methods of lock interface:

1. void lock():

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 exactly same behaviour of tradition synchronized keyword

2. boolean tryLock():

to acquire the lock without waiting. if the lock is available then the thread acquires that locks 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 in to waiting state

```
if(l.tryLock())
{
    Perform sage operations;
}
Else
{
    Perform alternative operations;
}
```

3. Boolean tryLock(long time, TimeUnit unit):

If lock is available then the thread will get the lock and continues its execution. If the lock is not available then 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 package

```
Enum TimeUnit
{
    NANOSECONDS;
    MICROSECONDS;
    MILLISECONDS;
    SECONDS;
    MINUTES;
    HOURS;
    DAYS;
}
```

Example:

If (l.tryLock(1000, TimeUnit.MILLISECONDS))

4. **Void lockInterruptibly()**

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 thread interrupted then thread won't get the lock

5. **Void unlock():**

To releases the lock. To call this method compulsory current thread should be owner of the lock otherwise we will get runtime exception saying "IllegalMonitorStateException".

ReentrantLock()

It is the implementation of lock interface and it is the direct child class of object.

Reentrant means a thread acquire same lock multiple times without any issue.

Internally reentrant lock increments thread personal count whenever we call lock method and decrements count value whenever thread calls unlock method and lock will be released whenever count reaches '0'.

Constructors:

1. **ReentrantLock l = new ReentrantLock();**: Creates an instance of ReentrantLock.
2. **ReentrantLock l = new ReentrantLock(Boolean fairness);**: creates reentrant lock with the given fairness policy. If the fairness is true the longest waiting thread can get the lock if it is available that is it follows "first come first policy". If fairness is false then which waiting thread will get chance we can't except.

Note: the default value for fairness is false.

Which of the following declarations are equal?

ReentrantLock l = new ReentrantLock(); → equal

ReentrantLock l = new ReentrantLock(true);

ReentrantLock l = new ReentrantLock(false); → equal

All the above

Important methods of ReentrantLock:

1. void lock()
2. boolean tryLock()
3. boolean tryLock(long t, TimeUnit t)
4. void lockInterruptibly()
5. void unlock()

6. **int getHoldCount()**: return number of holds on this lock by current thread.
7. **Boolean isHeldByCurrentThread()**: returns true if and only if lock is held by current thread
8. **Int getQueueLength()**: return 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()**: it 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 thread which acquires the lock.

Example:

```
import java.util.concurrent.locks.*;
class ReentrantLockOne
{
    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());//true
        l.unlock();
        System.out.println(l.isLocked());//false
        System.out.println(l.isFair());//false
    }
}
```

Example:

```
import java.util.concurrent.locks.*;
class Display
{
    ReentrantLock l = new ReentrantLock();
    public void wish(String name)
    {
        l.lock(); --→line one
        for (int i=0 ;i<10 ;i++ )
        {
            System.out.print("good morining: ");
            try
            {
                Thread.sleep(2000);
            }
            catch (InterruptedException e)
            { }
        }
    }
}
```

```

        System.out.println(name);
    }
    l.unlock(); →line two
}
}
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 ReentrantLock2
{
    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();
    }
}

```

If we comment line one and two, then threads will be executed simultaneously and we will get irregular output.

If we are not commenting lines one and two then the threads will be executed one by one and we will get regular output.

Demo program for tryLock() method:

```

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 sage operation");
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {
        }
        l.unlock();
    }
    else
    {
        System.out.println(Thread.currentThread().getName()+"....unable to
get lock and hence performing alternative operations");
    }
}
class ReentrantTryLockDemo
{
    public static void main(String[] args)
    {
        MyThread t1 = new MyThread("First Thread");
        MyThread t2 = new MyThread("second Thread");
        t1.start();
        t2.start();
    }
}

```

Output:

First Thread....got lock and performing sage operation
second Thread....unable to get lock and hence performing alternative operations

example 4:

```

import java.util.concurrent.locks.*;
import java.util.concurrent.*;
class MyThread extends Thread
{
    static ReentrantLock l = new ReentrantLock();
    MyThread(String name)
    {
        super(name);
    }

```

```

public void run()
{
    do
    {
        try
        {
            if (l.tryLock(5000, TimeUnit.MILLISECONDS))
            {

                System.out.println(Thread.currentThread().getName()+"....got lock");
                Thread.sleep(25000);
                l.unlock();

                System.out.println(Thread.currentThread().getName()+"..... releases lock");
                break;
            }
            else
            {

                System.out.println(Thread.currentThread().getName()+"..... unable to get lock and
will try again");
            }
        }
        catch (Exception e)
        {
        }
    }while(true);
}
}

class ReentrantTryLockTimeDemo4
{
    public static void main(String[] args)
    {
        MyThread t1 = new MyThread("First Thread");
        MyThread t2 = new MyThread("second Thread");
        t1.start();
        t2.start();
    }
}

```

Output:

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
First Thread..... releases lock
second Thread....got lock
second Thread..... releases lock

Thread Pools (Executor Framework)

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 over job.

Java 1.5 version introduces thread pool framework to implement thread pools.

Thread pool frame work is also known as executor framework.

We can create a thread pool as fallows.

```
ExecutorService service = Executors.newFixedThreadPool(3);
```

We can submit runnable job by using submit method.

```
service.submit(job);
```

we can shutdown Executor service by using shutdown() method

```
service.shutdown();
```

example:

```
import java.util.concurrent.*;
class PrintJob implements Runnable
{
    String name;
    PrintJob(String name)
    {
        this.name = name;
    }
    public void run()
    {
        System.out.println(name+".... job started by
Thread:"+Thread.currentThread().getName());
        try
        {
            Thread.sleep(5000);
        }
        catch (InterruptedException e)
        {
        }
        System.out.println(name+" ..... 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 reused for multiple jobs.

Output:

```

durga.... jab started by Thread:pool-1-thread-1
shiva.... jab started by Thread:pool-1-thread-3
ravi.... jab started by Thread:pool-1-thread-2
shiva ..... job compeleted by thread:pool-1-thread-3
durga ..... job compeleted by thread:pool-1-thread-1
suresh.... jab started by Thread:pool-1-thread-1
ravi ..... job compeleted by thread:pool-1-thread-2
pavan.... jab started by Thread:pool-1-thread-3
anil.... jab started by Thread:pool-1-thread-2
anil ..... job compeleted by thread:pool-1-thread-2
pavan ..... job compeleted by thread:pool-1-thread-3
suresh ..... job compeleted by thread:pool-1-thread-1

```

note:

while designing webservers and application servers we can use thread pool concept.

Callable and future:

In the case runnable job thread won't return any thing after completing the job.

If a thread a is required to return some result after execution then we should go for callable.

Callable interface contain only one method call()

Public object call() throws Exception

If we submit callable object to executor then after completing the job thread returns an object of the type Future.

That is Future object can be used to retrieve the result from callable job

Example:

```
import java.util.concurrent.*;
class MyCallable implements Callable
{
    int num;
    MyCallable(int num)
    {
        this.num = num;
    }
    public Object call() throws Exception
    {
        System.out.println(Thread.currentThread().getName()+" is .... responsible to
find sum of first "+num+"numbers");
        int sum=0;
        for (int i=1; i<=num;i++)
        {
            sum= sum+i;
        }
        return sum;
    }
}
class CallableFutureDemo
{
    public static void main(String[] args) throws Exception
    {
        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)
        {
            Future f = service.submit(job);
            System.out.println(f.get());
        }
        service.shutdown();
    }
}
```

Output:

```
pool-1-thread-1 is .... responsible to find sum of first 10numbers  
55  
pool-1-thread-2 is .... responsible to find sum of first 20numbers  
210  
pool-1-thread-3 is .... responsible to find sum of first 30numbers  
465  
pool-1-thread-1 is .... responsible to find sum of first 40numbers  
820  
pool-1-thread-2 is .... responsible to find sum of first 50numbers  
1275  
pool-1-thread-3 is .... responsible to find sum of first 60numbers  
1830
```

Difference between Runnable and Callable

Runnable	Callable
1. If a thread is not required to return any thing 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 contain only one method call()
3. Runnable job not required to any thing and hence return type of run() method is void.	3. Callable job is required to return something return type of call method is object .
4. With in the run method if there is any chance of raising checked Exception compulsory we should handle by using try- catch because we can't use throws keyword for run() method	4. With in call() method if there is any chance of checked Exception we are not required 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. Introduce 1.5 version

ThreadLocal

ThreadLocal class provide thread local variables.

ThreadLocal class maintains values per thread bases.

Each ThreadLocal object maintain a separate value like user id , transaction id etc., for each thread that access 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.

Example:

Consider a servlet which invokes some business methods.

We have a requirement to generate a unique transaction id for each and every request and we have to pass this transaction id to the business methods for this requirement we can use thread local to maintain a separate transaction id for every request that is every thread.

Note:

1. Thread local class introduce in 1.2 version and enhanced in 1.5 version
2. ThreadLocal can be associated with thread scope.
3. Total code which is executed by the thread has access to the corresponding thread local variables
4. A thread can access its own local variables and can't access other threads local variables.
5. Once thread entered in to dead state all its local variables are by default eligible for garbage collection.

Constructor:

ThreadLocal tl = new ThreadLocal()

Creates a thread local variable.

Methods:

1. **Object get()** : return the value of thread local variable associated with current thread
2. **Object initialValue()** : returns initial value of thread local variables associated with current thread. The default implementation of this method returns null. To customized 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 thread local variable associated with current thread. It is newly added method in 1.5 version after removal after trying to access it will be reinitialized once again by invoking its initial value method

Example:

```
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
    }
}
```

Example program:

```
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
withcustomer id :" +tl.get());
}
class ThreadLocalDemo2
{
    public static 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();
    }
}

```

Output:

Customer Thread-1 executing withcustomer id :1
 Customer Thread-2 executing withcustomer id :4
 Customer Thread-3 executing withcustomer id :3
 Customer Thread-4 executing withcustomer id :2

In the above program for every customer thread a separate customer id will be maintained by thread local object.

ThreadLocal v/s Inheritance

Parent threads thread local 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 thread values exactly same as parents threads value but we can provide the customized value for child thread by overriding childValue() method.

Constructor:

```
InheritableThreadLocal tl = new InheritableThreadLocal();
```

Methods:

InheritableThreadLocal is the child class thread local and hence all method present in thread local by default available to inheritable thread local.

In addition this methods it contain only one method

Public Object childValue(Object parentValue)

Example:

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 value: "+tl.get());  
        ChildThread ct = new ChildThread();  
        ct.start();  
    }  
}  
class ChildThread extends Thread  
{  
    public void run()  
    {  
        System.out.println("child value:"+ParentThread.tl.get());  
    }  
}  
class ThreadLocalDemo3  
{  
    public static void main(String[] args)  
    {  
        ParentThread pt = new ParentThread();  
        pt.start();  
    }  
}
```

In the above program if we replace InheritableThreadLocal with ThreadLocal and if we are not overriding childValue method then the out put is

Parent thread value — pp
Child thread value—null

In the above program if we are maintain InheritableThreadLocal and if we are not overriding childValue method

Parent thread value — pp
Child thread value — pp

Inner Classes

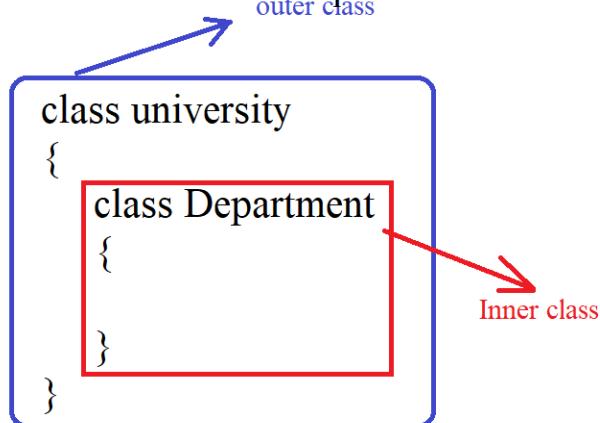
Some times we can declare a class inside another class such type classes are called inner classes.

Inner classes concept introduced in 1.1 version to fix GUI bugs as the part of event handling but because of powerful future and benefits of inner class slowly programmer are 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

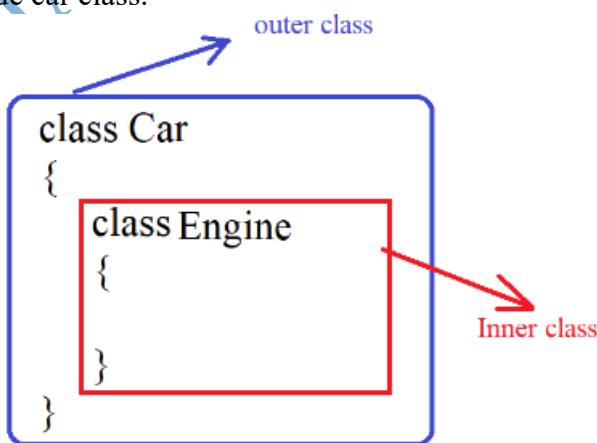
Example 1:

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



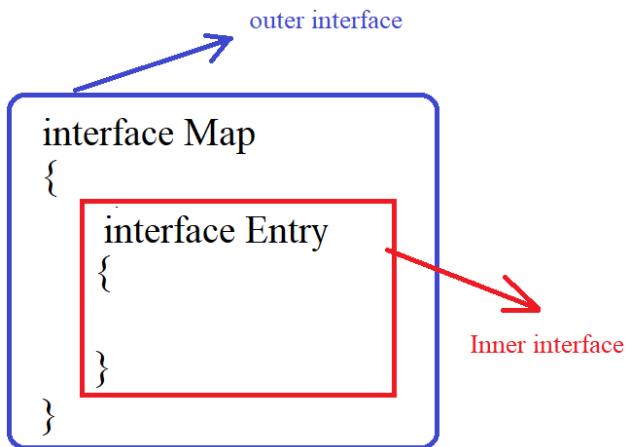
Example 2:

Without existing car object there is no chance of existing engine object hence we have to declare engine class inside car class.



Example 3:

Map is a group of key value pairs and each key value pair is called an entry. With out existing map object there is no chance of existing entry object. Hence interface entry is define inside map interface.



Note:

1. Without existing outer class object there is no chance of existing inner class object
2. The relation between outer class and inner class is not is-a relation and it is has-a relationship (composition or aggregation).

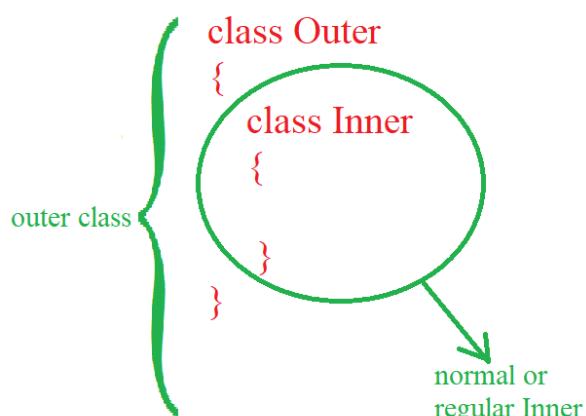
Based on position of declaration and behaviour all inner classes divided into 4 types

1. Normal or regular inner classes
2. Method local inner classes
3. Anonymous inner classes
4. Static nested classes

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.

Example 1:

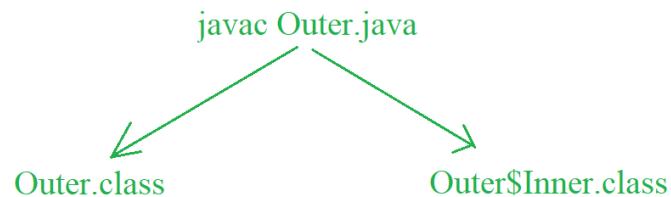


Example 2:

```

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

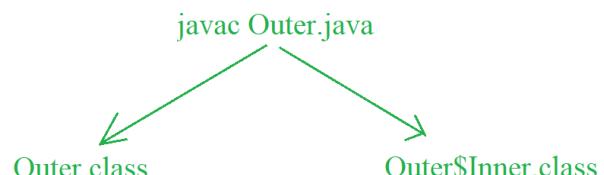
```



```

java Outer
RE:NoSuchMethodError: main
java Outer$Inner
RE: NoSuchMethodError: main

```



```

java Outer
o/p: Outer class main method
java Outer$Inner
RE: NoSuchMethodError: main

```

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

Example:

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

CE: Inner classes cannot have static declarations

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");
        }
    }
    public static void main(String[] args)
    {
        New Outer().  
new Inner().  
m1();  
    }
}
```

Outer.Inner I = new Outer().newInner()

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");
        }
    }
}
Class Test
{
    public static void main(String[] args)
    {
        Outer o = new Outer();
        Outer.Inner i = o.new Inner();
        i.m1();
    }
}

```

Summary:

accessing Inner calls code

from static area of outer class
 or
form 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 member of outer class directly

```
class Outer
{
    int x =10;
    static int y = 20;
    class Inner
    {
        public void m1()
        {
            System.out.println(x); // 10
            System.out.println(y); //20
        }
    }
}
class OuterDemo
{
    public static void main(String[] args)
    {
        new Outer().new Inner().m1();
    }
}
```

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

```
class Outer
{
    int x =10;
    class Inner1
    {
        int x =100;
        public void m1()
        {
            int x =1000;
            System.out.println(x);//1000
            System.out.println(this.x);//100
            System.out.println(Inner.this.x);//100
            System.out.println(Outer.this.x);//10
        }
    }
    public static void main(String[] args)
    {
        new Outer1().new Inner1().m1();
    }
}
```

The only applicable modifier for outer classes are

public
<default>
final
abstract
strictfp

But for inner classes applicable modifiers are

public
<default>
final
abstract
strictfp
Private
Protected
static

nesting of inner classes:

inside inner class we can declare another inner class that is nesting of inner classes is possible.

```
class A
{
    class B
    {
        class C
        {
            public void m1()
            {
                System.out.println("innerMost class Method");
            }
        }
    }
}

class Test360
{
    public static void main(String[] args)
    {
        A a=new A();
        A.B b =a.new B();
        A.B.C c = b.new C();
        c.m1();

        new A().new B().new C().m1();
    }
}
```

Method local Inner classes:

Some times we can declare a class inside a method such type inner classes are called method local inner classes.

The main purpose of method local inner class is to define method specific repeated required functionality.

Method local inner classes are best suitable to meet nested method requirements.

We can access method local inner classes only with in the method where we declare. 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.

```
class Test
{
    public void m1()
    {
        class Inner
        {
            public void sum(int x,int y)
            {
                System.out.println("the sum: "+(x+y));
            }
        }
        Inner i = new Inner();
        i.sum(10,20);
        i.sum(100,200);
        i.sum(1000,2000);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}
Oupt put:
the sum: 30
the sum: 300
the sum: 3000
```

we can declare method local inner class inside both instants and static methods.

If we declare inner class inside instance method then from that method local inner class we can access both static and non- static member 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.

```

class Test
{
    int x =10;
    static int y=20;
    public void m1()
    {
        class Inner
        {
            public void m2()
            {
                System.out.println(x); //10→ line one
                System.out.println(y); //20
            }
        }
        Inner i = new Inner();
        i.m2();
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}

```

If we declare m1() method as static then at line one we will get compile time error saying “non-static variable ‘x’ cannot be reference from a static context”

**** from method local inner class we can access local variable of the method in which we declare inner class. If the local variable declared as final then we can access.

Example:

```

class Test7
{
    public void m1()
    {
        int x=10;
        class Inner
        {
            public void m2()
            {
                System.out.println(x);
            }
        }
        Inner i = new Inner();
        i.m2();
    }
    public static void main(String[] args)
    {
        Test7 t = new Test7();
        t.m1();
    }
}

```

According to 1.6 version we get CE: local class accessed from within inner class; needs to declare final

If we declare x as final then we won't get any compile time error.

Consider the following code:

```
class Test8
{
    int i=10;
    static int j = 20
    public void m1()
    {
        int k =30;
        final int m = 40;
        class Inner
        {
            public void m2()
            {
                Line one;
            }
        }
    }
}
```

At line one which of the following variables we can access directly

- I → valid
- J → valid
- K → invalid
- M → valid

If we declare m1() method as static then at line one which variables we can access directly

- I → invalid
- J → valid
- K → valid
- M → valid

If we declare m2() method as static then at line one which variables we can access directly

- I
- J
- K
- m

Ans: we will get compile time error we can't declare static members inside inner classes

The only applicable modifiers method local inner classes are **final, abstract, strictfp**. If we are trying to apply any other modifier then we will get compile time error

Anonymous Inner Classes

Some time 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)

Based on declaration and behaviour there are 3 types of inner classes.

1. Anonymous inner class that extends a class
2. Anonymous inner class the implements an interface
3. Anonymous inner class that defined inside arguments

Anonymous Inner class that extends a class:

```
class PopCorn
{
    public void taste()
    {
        System.out.println("salty");
    }
    //100 more methods
}
class Test
{
    public static void main(String[] args)
    {
        PopCorn p = new PopCorn()
        {
            public void taste()
            {
                System.out.println("spicy");
            };
            p.taste();
        PopCorn p1 = new PopCorn();
        p1.taste();
        PopCorn p2 = new PopCorn()
        {
            public void taste()
            {
                System.out.println("sweet");
            };
            p2.taste();
        System.out.println(p.getClass().getName());
        System.out.println(p1.getClass().getName());
        System.out.println(p2.getClass().getName());
    }
}
```

→ spicy
→ salty
→ sweet
→ Test\$1
→ PopCorn
→ Test\$2

The generated .class files are

1. PopCorn for PopCorn.class
2. Test class for Test.class
3. First Anonymous class for Test\$1.class
4. Second Anonymous class for Test\$2.class

Analysis:

1. PopCorn p = new PopCorn(): just we are creating PopCorn object.

2. PopCorn p = new PopCorn()

```
{  
};
```

i) We are declaring 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()
```

```
{
```

```
    System.out.println("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

```
class MyThread extends Thread{  
    public void run(){  
        for (int i=0;i<10 ;i++ )  
        {  
            System.out.println("child thread");  
        }  
    }  
}  
class NormalApproach  
{  
    public static void main(String[] args)  
    {  
        MyThread t= new MyThread();  
        t.start();  
        for (int i=0;i<10 ;i++ )  
        {  
            System.out.println("main thread");  
        }  
    }  
}
```

anonymous Inner class approach

```
class AnonymousApproach
{
    public static void main(String[] args)
    {
        Thread t= new Thread()
        {
            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");
        }
    }
}
```

Anonymous Inner class that implements an interface

Defining a thread by implementing runnable Interface

normal class approach for runnable interface

```
class MyRunnable implements Runnable{
    public void run(){
        for (int i=0;i<10 ;i++ )
        {
            System.out.println("child thread");
        }
    }
}
class NormalApproachInterface
{
    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 class approach for runnable interface

```
class AnonymousApproachInterface
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            public void run()
            {
                for (int i=0;i<10 ;i++ )
                {
                    System.out.println("child thread1");
                }
            };
        Thread t = new Thread(r);
        t.start();
        for (int i=0;i<10 ;i++ )
        {
            System.out.println("main thread1");
        }
    }
}
```

Anonymous inner class that defines inside arguments:

```
class AnonymousVariable
{
    public static void main(String[] args)
    {
        new Thread(new Runnable(){
            public void run()
            {
                for (int i = 0;i<10 ;i++ )
                {
                    System.out.println("child thread");
                }
            }).start();
        for (int i = 0;i<10 ;i++ )
        {
            System.out.println("main thread");
        }
    }
}
```

Normal java class v/s anonymous Inner Class

1. A normal java class can extend only one class at a time off course 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 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 simultaneously.
4. In normal java class we can write any number of constructors. But in anonymous inner class we can't write any constructor explicitly (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 time 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

Where anonymous inner classes are best suitable?

We can use anonymous inner classes frequently in GUI based application to implement event handling.

```
class MyGUIFrame extends JFrame
{
    JButton b1,b2,b3,b4,b5,b6;
    .....
    b1.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            //b1 specific Functionallity
        }
    });
    b2.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            //b2 specific Functionallity
        }
    });
}
```



Static Nested Classes

Some times we can declare inner class with static modifier such type inner classes are called static nested classes.

In the case normal or regular inner class without existing outer class object there is no chance of existing inner class object that is inner class object is strongly associated with outer class object.

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.

```
//static nested class demo
class StaticNestedDemo
{
    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();
    }
}
```

If we want to create nested class object from outside of outer class then we can create as follows.

```
Outer.Nested n = new Outer.Nested();
```

In normal or regular classes we can't declare any static members

But in static nested classes we can declare static members including main method hence we can invoke static nested class directly from command prompt

```
//static nested class demo
class StaticNestedDemo2
{
    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");
}
}

```

Ouput:

```

java StaticNestedDemo2
outer class main method

```

```

java StaticNestedDemo2$Nested
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

```

//static nested class demo
class StaticNestedDemo3
{
    int x =10;
    static int y = 20;
    static class Nested
    {
        public void m1()
        {
            System.out.println(x);
            System.out.println(x);
        }
    }
}//CE:non-static variable x cannot be referenced from a static context

```

Difference between normal and regular inner class and static nested class:

Normal or regular inner class	Static nested classes
1. Without existing outer class object there is no chance of existing inner class object that is inner class object strongly associated with outer class object	1. Without existing outer class object there may be a chance of existing static nested class object hence that is static nested class object is not strongly associated with outer class object.
2. In normal or regular inner classes 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 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 object then we can declare class inside a class

Example:

University consists 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 implementation of an interface and all this implements are related a particular class then we can define interface inside a class.

```
class VechiclTypes
{
    interface vehicle
    {
        public int getNoOfWheels();
    }
    class Bus Implements vehicle
    {
        public int getNoOfWheel()
        {
            return 6;
        }
    }
    Class Auto implements vechicle
    {
        Public int getNoOfWheels()
        {
            Return 3;
        }
    }
}
```

Case 3: Interface inside interface

We can declare interface inside interface.

Example:

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 interface

interface map

```
{  
    interface Entry  
    {  
    }  
}
```

102	Ravi
103	Silva
104	Davey

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 implement inner interface that is we can implement outer and inner interface independently.

```
interface Outer3  
{  
    public void m1();  
    interface Inner3  
    {  
        public void m2();  
    }  
}  
class Test10 implements Outer3  
{  
    public void m1()  
    {  
        System.out.println("outer interface method implementation");  
    }  
}  
class Test11 implements Outer3.Inner3  
{  
    public void m2()  
    {  
        System.out.println("Inner interface method implementation");  
    }  
    public static void main(String[] args)  
    {  
        Test11 t = new Test11();  
        t.m2();  
        Test10 t1 = new Test10();  
        t1.m1();  
    }  
}
```

Case 4: Class inside interface

If functionality of a class is closely associated with interface then it is highly recommended to declare a class inside interface

Example:

```
interface EmailService
{
    public void sendMail(EmailDetails e);
    class EmailDetails
    {
        String toList;
        String ccList;
        String subject;
        String body;
    }
}
```

In the above example email details is required only for email service and we are not using anywhere else hence email details class is recommended to declare inside email service interface.

We can also define a class inside interface to provide default implementation for that interface.

```
interface Vehicle
{
    public int getNumberOfWheels();
    class DefaultVehicle implements Vehicle
    {
        public int getNumberOfWheels()
        {
            return 2;
        }
    }
    class Bus implements Vehicle
    {
        public int getNumberOfWheels()
        {
            return 6;
        }
    }
}
```

```

class Test13
{
    public static void main(String[] args)
    {
        Vehicle.DefaultVehicle 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 the default implementation of vehicle interface whereas Bus Is customized implementation of vehicle interface.

Note:

The class which is declared inside interface is always public static whether we are declaring or not hence we can create class object directly with out having outer interface object

Conclusions:

Among classes and interface we can declare any thing inside any thing

Class A	Class A	Interface A	Interface A
{	{	{	{
Class B	Interface B	Interface B	Class B
{	{	{	{
}	}	}	}
Valid	Valid(static)	Valid(public & static)	Valid(public & static)

the interface which is declared inside interface is always public and static whether we are declaring or not.

The class which is declared inside interface is always public and static whether we are declaring or not.

The interface which is declared inside a class is always static but need not be public.

Java.lang package

1. introduction
2. object class
3. String class
4. StringBuffer class
5. StringBuilder class
6. Wrapper classes
7. Autoboxing & auto un-boxing

For writing any java program whether it is simple or complex the most commonly required classes and interfaces are grouped into a separated 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 to every java program.

Object class:

The most commonly required methods for every java class (whether it is predefined class or customized class) or defined in separate class which is nothing but object class.

Every class in java is the child class of object either directly or indirectly so that object class method by default available to every java class.

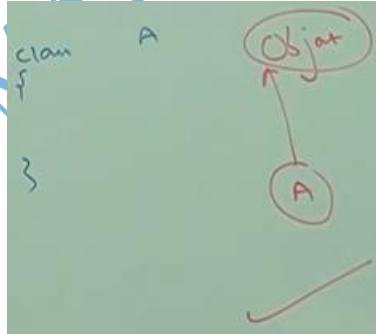
Hence object class is considered as root of all java classes

Note:

1. If our class doesn't extend any other class then only our class is the direct child class of object

Example:

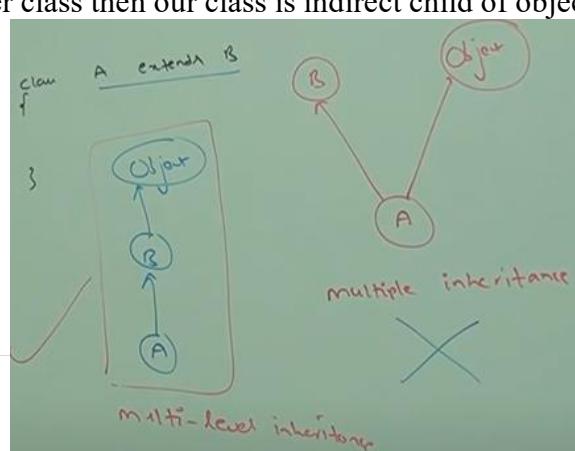
```
Class A
{
}
```



2. If our class extends any other class then our class is indirect child of object

Example

```
Class A extends B
{}
```



3. Either directly or indirectly java won't provide support for multiple inheritance with respect to classes .

**** Object class defines the following 11 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 native 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()

12. 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.

toString():

we can use toString() method to get string representation of an object

String s = obj.toString();

When ever we are trying to print object reference internally toString() method will be called.

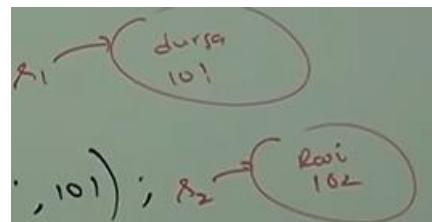
Example:

```
Student s = new Student();
System.out.println(s); → s.toString();
```

If our class doesn't contain toString() method then object class toString() method will be executed.

Example:

```
class StudentObject
{
    String name;
    int rollno;
    StudentObject(String name, int rollno)
    {
        this.name = name;
        this.rollno = rollno;
    }
}
```



```

public static void main(String[] args)
{
    StudentObject s1 = new StudentObject("durga",101);
    StudentObject s2 = new StudentObject("ravi",102);
    System.out.println(s1);           //StudentObject@15db9742
    System.out.println(s1.toString()); //StudentObject@15db9742
    System.out.println(s2);           //StudentObject@6d06d69c
}

```

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 trying to print student object reference to print his name and rollno we have to over ride `toString()` method as follows

```

public String toString()
{
    return name+"....."+rollno;
    //return "this is student with the name:"+name+"and rollno:"+rollno;
}

```

In all wrapper class, in all collection classes, string class, `StringBuffer`, `StringBuilder` classes `toString()` method overridden for meaning full string representation hence it is highly recommended to over ride `toString` method in our class also.

Example:

```

import java.util.*;
class Test5
{
    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]
    Test5 t = new Test5();
    System.out.println(t);          //Test5
}

```

hashCode ():

for every object a unique number generated by JVM which is nothing but hashCode(). hashCode() won't represent address of object. JVM will use hashCode() while saving objects into Hashing related data structure like HashTable, HashMap, HashSet etc., the main advantage of saving objects base on hashCode() is search operation will become easy (the most power full search algorithm up to day is Hashing).

If we are giving the chance to object class hashCode() method it will generate hashCode based on address of the object. It doesn't mean hash code represents address. Based on our requirement we can override hashCode() method in our class to generates our own hashCode.

Overriding hashCode() method is said to be proper if and only if for every object we have to generate a unique number as hashCode.

```

Class Student
{
    .....
    .....
    Public int hashCode()
    {
        Return 100;
    }
} improper way

```

This is improper way of overriding hashCode() method because for all student object we are generating same number as hashCode.

```

Class Student
{
    .....
    .....
    Public int hashCode()
    {
        Return rollno;
    }
}

```

```
    }  
} proper way
```

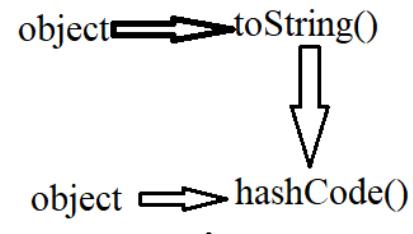
This is proper way of overriding hashCode() method because we are generating a different hashCode() for every object.

toString() v/s hashCode():

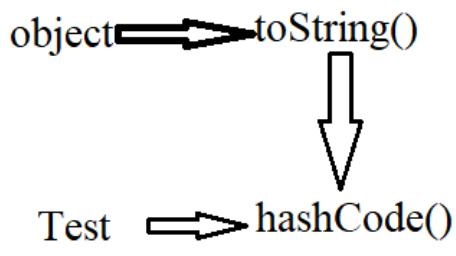
If we are giving chance to object class toString() method it will internally calls hashCode() method.

If we overriding toString() method then our toString() method may not call hashCode() method.

```
class Test14  
{  
    int i;  
    Test14(int i)  
    {  
        this. i= i;  
    }  
    public static void main(String[] args)  
    {  
        Test14 t1 = new Test14(10);  
        Test14 t2= new Test14(100);  
  
        System.out.println(t1);//Test14@15db9742  
        System.out.println(t2);//Test14@6d06d69c  
    }  
}
```



```
class Test14A  
{  
    int i;  
    Test14A(int i)  
    {  
        this. i= i;  
    }  
    public int hashCode()  
    {  
        return i;  
    }  
    public static void main(String[] args)  
    {  
        Test14A t1 = new Test14A(10);//Test14A@a  
        Test14A t2= new Test14A(100);//Test14A@64  
        System.out.println(t1);  
        System.out.println(t2);  
    }  
}
```



```

class Test14B
{
    int i;
    Test14B(int i)
    {
        this.i = i;
    }
    public String toString()
    {
        return i + " ";
    }
    public int hashCode()
    {
        return i;
    }
    public static void main(String[] args)
    {
        Test14B t1 = new Test14B(10); //10
        Test14B t2 = new Test14B(100); //100

        System.out.println(t1);
        System.out.println(t2);
    }
}

```

Test toString()

equals() method

we can use equals() method to check equality of two objects.

Example:

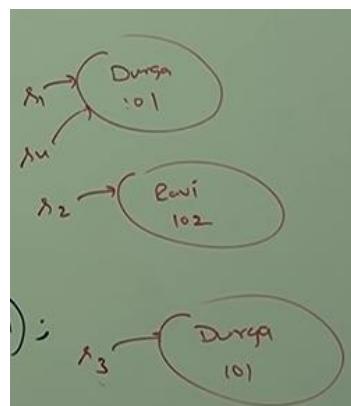
obj1.equals(obj2)

if our class doesn't contain equals() method the object class equals() method will be executed.

```

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);
        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
    }
}

```

In the above example object class equals() method got executed which is meant for reference comparison (address comparison) that is if two references point to the same object then only .equals() method returns true.

Based on our requirement we can override equals method for content comparison.

While overriding equals() method for content comparison we have to take care about the following .

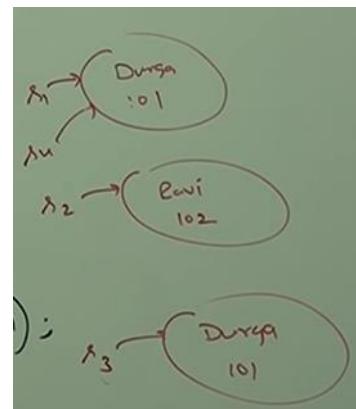
1. What is the meaning equality(that is whether we have to check only names or only rollno or both)
2. We are passing different type of object over equals() method should not raise classCast Exception that is we have to handle ClassCastException to return false.
3. If we are passing null argument then our equals() method should not raise NullPointerException that is we have to handle NullPointerException to return false.

The following is the proper way of overriding equals() method for student class content comparison

```

S1.equals(s2) → this
public boolean equals(Object obj)
{
    try
    {
        String name1 = this.name;
        int rollNo1 = this.rollno;
        StudentOne s = (StudentOne) obj;      →RE:CCE
        String name2 = s.name;                →RE:NPE
        int rollNo2= s.rollno;               →RE:NPE
        if(name1.equals(name2)&& rollNo1==rollNo2)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    catch (ClassCastException e)
    {
        return false;
    }
    catch(NullPointerException e)
    {
        return false;
    }
}

```



```

    }
StudentOne s1 = new StudentOne("durga",101);
StudentOne s2 = new StudentOne("ravi",102);
StudentOne s3 = new StudentOne("durga",101);
StudentOne s4 = s1;
System.out.println(s1.equals(s2));//false
System.out.println(s1.equals(s3));//true
System.out.println(s1.equals(s4));//true
System.out.println(s1.equals("durga")); //false
System.out.println(s1.equals(null)); //false

```

Simplified version of equals() method:

```

public boolean equals(Object obj)
{
    try
    {
        StudentOne s = (StudentOne) obj;
        String name2 = s.name;
        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 equals() method

```

public boolean equals(Object obj)
{
    if(obj instanceof StudentOne)
    {
        StudentOne s = (StudentOne) 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 reference pointing to the same object then without performing any comparison .equals() method returns true directly.

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

In string class .equals() method overridden for content comparison hence, even though objects are different if content is same then .equals method true.

```
StringBuffer s1 = new StringBuffer("durga");
StringBuffer s2 = new StringBuffer("durga");
System.out.println(s1 == s2); //false
System.out.println(s1.equals(s2));// 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.

getClass()

we can use getClass() method to get runtime class definition of an object.

Public final Class getClass()

By using Class Class object we can access class level properties like fully qualified name of the class Methods information, constructors information etc.,

```
import java.lang.reflect.*;
class Test15
{
    public static void main(String[] args)
    {
        int count = 0;
        Object o = new String("durga");
```

```

Class c = o.getClass();
System.out.println("fully qualified name of 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: "+count);
}
}

```

Example 2:

To display database vendor specific connection interface implemented class name.

```

Connection con = DriverManager.getConnection(...);
System.out.println(con.getClass().getName());

```

Note:

1. After loading every .class file JVM will create an object of the type java.lang.Class in the heap area. Programmer can use this class object to get class level information.
2. We can use getClass() method very frequently in reflections.

Finalize()

Just before destroying object garbage collector calls finalize() method to perform clean up activities. Once Finalize() method completes automatically garbage collector destroy the object.

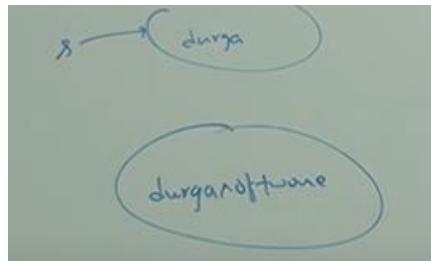
Wait(), notify(), notifyAll()

We can use these methods for inter thread 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.

Java.lang.String

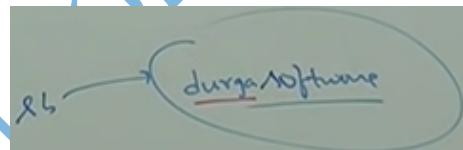
Case 1:

```
String s = new String("durga");
s.concat("software");
System.out.println(s); //durga
```



Once we creates string object we can't perform any change 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

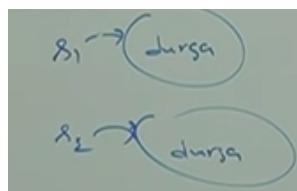
```
StringBuffer sb = new StringBuffer("durga");
sb.append("software");
System.out.println(sb); // durgasoftware
```



Once we create StringBuffer object we can perform any change in the existing object this changeable behaviour is nothing but mutability of string buffer object.

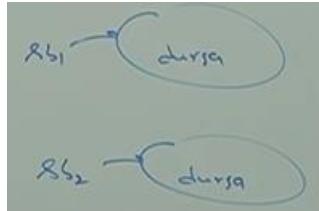
Case:2

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



In string class .equals() method is overridden for content comparison. Hence even though objects are different if content is same .equals() method returns true.

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

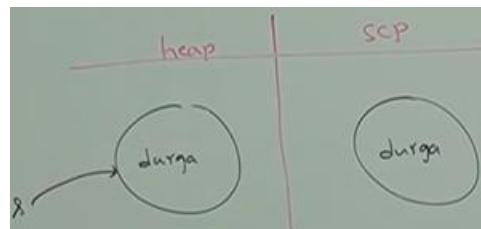


In string buffer class .equals() method is not overridden of content comparison. Hence object class .equals() method got executed which is meant for reference comparison(address comparison) due to this if objects are different .equals() method returns false even though content is same.

Case: 3

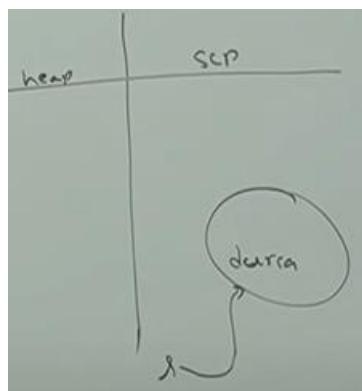
String s = new String("durga");

In this case two objects will be created one in the heap area and other is 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.

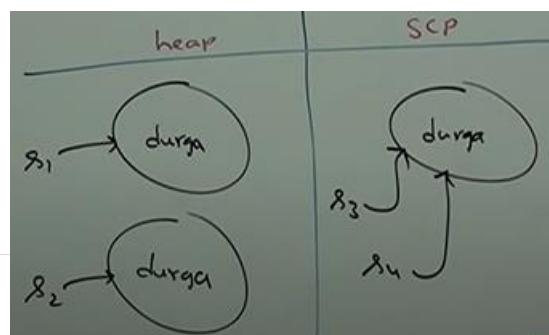


Note:

1. Object creation in SCP is optional. First it will check if there is any object already present in SCP with required content. If 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 for SCP but not for the heap.
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 object will be destroyed automatically at the time JVM shutdown.

Example 2:

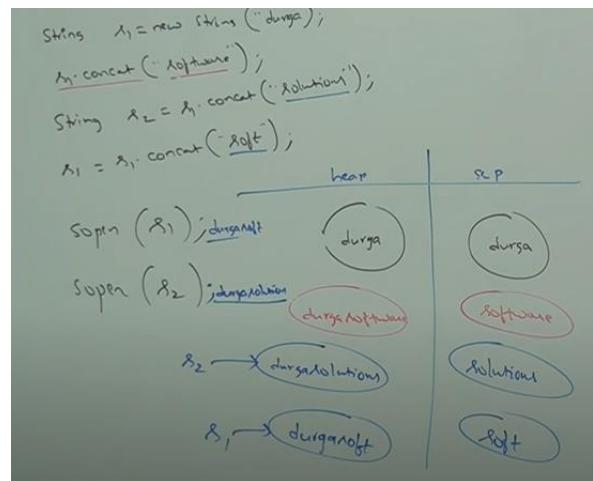
```
String s1 = new String("durga");
String s2 = new String("durga");
String s3 = "durga";
String s4 = "durga";
```



Whenever we are using new operator compulsory a new object will be created in the heap area. Hence there may be a chance existing two objects with same content in the heap area but not in SCP that is duplicate objects are possible in the heap area but not in SCP.

Example:3

```
String s1 = new String("durga");
s1.concat("software");
String s2 = s1.concat("solutions");
s1 = s1.concat("Soft");
System.out.println(s1); //durgasoft
System.out.println(s2); //durgasolutions
```

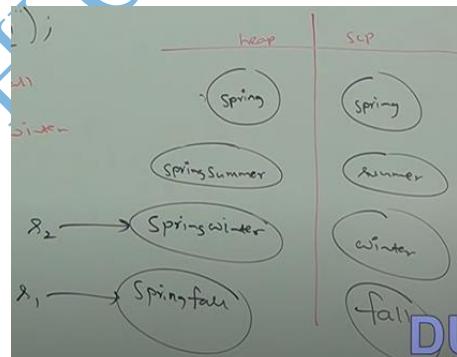


Note:

1. For every string constant one object will be placed in SCP area.
2. Because of some run time operations if object is required to create that object will be placed only in the heap area but not in SCP area.

Example 4:

```
String s1 = new String("spring");
s1.concat("summer");
String s2 = s1.concat("winter");
s1 = s1.concat("fall");
System.out.println(s1); //springfall
System.out.println(s2); //springwinter
```



Constructors of String class

1. `String s = new String();`
Creates any empty string object
2. `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 String buffer
4. `String s = new String (char[] ch);`
Creates an equivalent string object for the given char array

```
Examplt: char[] ch = {'a', 'b', 'c', 'd'}  
String s = new String(ch);  
System.out.println(s); //abcd
```

5. **String s = new String(byte[] b);**

Creates an equivalent string object for the given byte array

```
Example: byte[] b = {100,101,102,103}  
String s = new String(b);  
System.out.println(s); // defg
```

Important methods of String class:

1. **Public char charAt(int index);**

Returns the character locating at specified index.

Example:

```
String s = "durga"  
System.out.println(s.charAt(3)); //g  
System.out.println(s.charAt(30)); //  
RE: StringIndexOutOfBoundsException
```

2. **Public String concat(String s)**

The overloaded + and += operators also meant for concatenation only

Example:

```
String s = "durga";  
s = s.concat("software");  
// s = s + "software";  
// s += "software";  
System.out.println(s); // durgasoftware
```

3. **Public Boolean equal(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.

Example:

```
String s = "java";  
System.out.println(s.equals("JAVA")); //FALSE  
System.out.println(s.equalsIgnoreCase("JAVA")); //true
```

Note:

In general we can use equalsIgnoreCase() method to validate usernames where case is not important whereas 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

Example:

```
String s = "abcdefg";
System.out.println(s.substring(3)); //defg
System.out.println(s.substring(2,6)); //cdef
```

7. public int length();

Return number of characters present in the String

Example:

```
String s = "durga";
System.out.println(s.length()); //CE: cannot find symbol;
                                Symbol: variable length;
                                Location :java.lang.String
System.out.println(s.length()); //5
```

Note:

length variable applicable for arrays but not for string objects. Whereas length() method applicable for string objects but not for arrays.

8. public String replace(char oldch, char newCh);

eg: String s = "ababa";

```
System.out.println(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);

Return 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
```

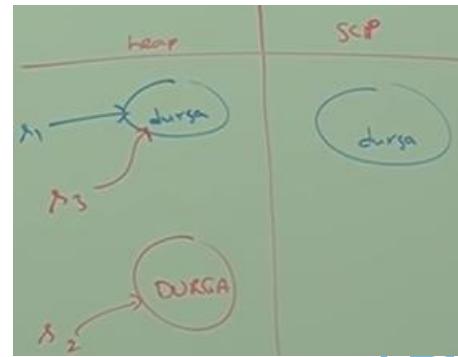
Note:

*** because of runtime operation if there is a change in the content then with those changes a new object will be created in the heap. If there is no change in the content existing object will be reused and new object won't be created.

Whether the object present in heap or SCP the rule is same.

Example:

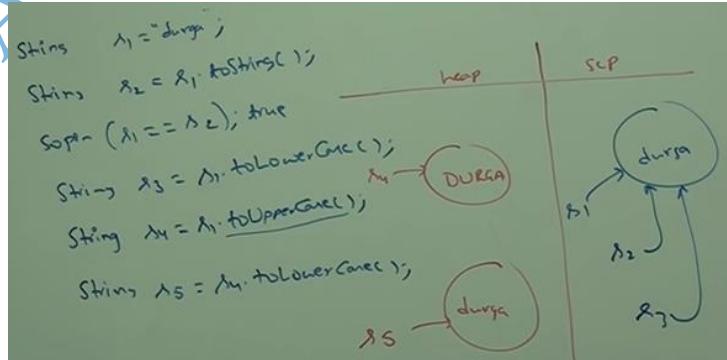
```
class StringMethod
{
    public static void main(String[] args)
    {
        String s1 = new String("durga");
        String s2 = s1.toUpperCase();
        String s3= s1.toLowerCase();
        System.out.println(s1==s2);
        System.out.println(s1==s3);
    }
}
```



```
class StringMethod
{
    public static void main(String[] args)
    {
        String s1 = new String("durga");
        String s2 = s1.toUpperCase();
        String s3= s1.toLowerCase();
        System.out.println(s1==s2);
        System.out.println(s1==s3);
        String s4= s2.toLowerCase();
        String s5 = s4.toUpperCase();
    }
}
```



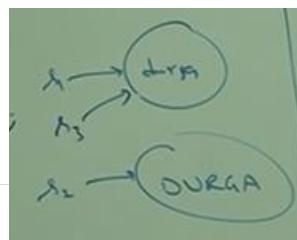
```
String s1 = "durga";
String s2 = s1.toString();
System.out.println(s1==s2);
String s3= s1.toLowerCase();
String s4= s1.toLowerCase();
String s5 = s4.toLowerCase();
```



How to create our own immutable class

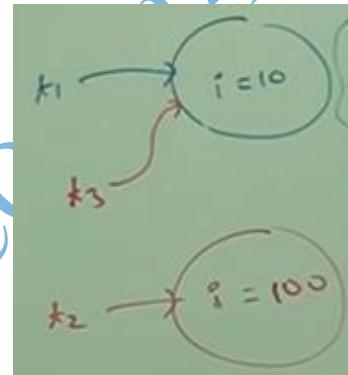
Once we creates an object we can't perform any changes in that object. If we are trying to perform any change and if there is a change in the content then with those changes a new object will be created. If there is no change in the content existing object will be reused this behaviour is nothing but immutability.

```
String s1 = new String("durga");
String s2 = s1.toUpperCase();
String s3= s1.toLowerCase();
```



We can create our own immutable class

```
Final public class Test
{
    Private int I;
    Test(int i)
    {
        This.i = I;
    }
    Public Test modify(int i)
    {
        If (this.i == i)
        {
            Return this;
        }
        Else
        {
            Return (new Test(i));
        }
    }
    .....
}
Test t1 = new Test(10);
Test t2 = t1.modify(100);
Test t3 = t1.modify(10);
System.out.println(t1 == t2); //false
System.out.println(t1 == t3); //true
```



Once we create a Test object we can't perform any change in the existing 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 created and if there is no change in the content then existing object will be reused.

Final v/s immutability

Final applicable for variables but not for objects whereas immutability applicable for objects but not for variables.

By declaring a reference variable as final we won't get any immutability nature even though reference variable is final we can perform any type of change in the corresponding object. But we can't perform reassignment for that variable.

Hence final and immutable both are different concepts

Example:

```
final StringBuffer sb = new StringBuffer("durga");
sb.append("software");
System.out.println(sb); //durgasoftware
sb = new StringBuffer("solutions"); //CE: cannot assign a value to final variable sb
```



which of the following is meaning full.

- | | |
|--------------------|-----------|
| Final variable | → valid |
| Immutable variable | → invalid |
| Final object | → invalid |
| Immutable object | → valid |

StringBuffer

If the content is fixed won't change frequently 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 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 StringBuffer reaches its max capacity a new StringBuffer object will be created with

$$\text{new capacity} = (\text{current capacity}+1) * 2$$

example:

```
StringBuffer sb = new StringBuffer();
System.out.println(sb.capacity()); //16
sb.append("abcdefghijklmnp");
System.out.println(sb.capacity()); //16
sb.append("q");
System.out.println(sb.capacity()); //34
```

2. `StringBuffer sb = new StringBuffer(int intialcapacity);`

Creates an empty StringBuffer object with specified initial capacity.

3. `StringBuffer sb = new StringBuffer(String s);`

Creates an equivalent String Buffer for the given string with
Capacity = s.length() + 16

Example:

```
StringBuffer sb = new StringBuffer("durga");
System.out.println(Sb.capacity()); //16+5=21
```

Important method of StrinBuffer

1. Public int length();
2. Public int capacity();
3. Public char charAt(int index);

Example:

```
StringBuffer sb = new StringBuffer("durga");
System.out.println(sb.charAt(3)); //g
System.out.println(sb.charAt(30)); // RE: StringIndexOutOfBoundsException
```

4. Public void setCharAt(int index, char ch);
To replace the character located at specified index with provide character

5. Public StringBuffer append(String s);
Public StringBuffer append(int i);
Public StringBuffer append(long l);
Public StringBuffer append(double d);
Public StringBuffer append(char c);
Public StringBuffer append(boolean b);
Public StringBuffer append(float f);

Example:

```
StringBuffer sb = new StringBuffer();  
Sb.append("pi value is : ");  
Sb.append(3.14);  
Sb.append(" it si exactly : ");  
Sb.append(true);  
System.out.println(sb);
```

6. Public String insert(int index, String s)
Public String insert(int index, int i)
Public String insert(int index, double d)
Public String insert(int index, char ch)
Public String insert(int index, Boolean b)

Example:

```
StringBuffer sb = new StringBuffer("abcdefg");  
Sb.insert(2, "xyz");  
System.out.println(sb); // abxyzcdefgh
```

7. Public StringBuffer delete(int begin, int end)
To delete character located from begin index to end-1

8. Public StringBuffer deleteCharAt(ind index)
To delete the character located at specified index

9. Public StringBuffer reverse();
Example:

```
StringBuffer sb = new StringBuffer("durga");  
System.out.println(sb.reverse()); //agrud
```

10. Public void setLength(int length());
Example:
StringBuffer sb = new StringBuffer("aiswaryaabhi");
Sb.setLength(8);
System.out.println(sb); //aiswarya

11. Public void ensureCapacity(int capacity);
To increase capacity on fly based on our requirement.
Example:

```
StringBuffer sb = new StringBuffer();
System.out.println(sb.capacity()); //16
sb.ensureCapacity(1000);
System.out.println(sb.capacity()); //1000
```

12. Public void trimToSize();
To deallocate extra allocated free memory
Example:

```
StringBuffer sb = new StringBuffer(1000);
sb.append("abc");
sb.trimToSize();
System.out.println(sb.capacity()); //3
```

StringBuilder

Every method present in StringBuffer is synchronized and hence only one thread is allowed to operate on StringBuffer object at time which makes creates performance problem. To handle this requirement sun people introduced StringBuilder concept in 1.5 version.

StringBuilder is exactly same as StringBuffer except the following difference.

StringBuffer	StringBuilder
1. Every method present in StringBuffer is synchronized	1. Every method present in StringBuilder is non-synchronized
2. At a time only one thread is allow to operate on string buffer object and hence StringBuffer object is thread safe	2. At a time multiple thread are operate on StringBuilder object and hence StringBuilder is not thread safe
3. Threads are required to wait to operate on StringBuffer object and hence relatively performance is low	3. Threads are not required to wait to operate on StringBuilder object and hence relatively performance is high
4. Introduced in 1.0 version	4. Introduce in 1.5 version.

Note:

Except the above difference every thing is same in StringBuffer and StringBuilder (including methods and constructors);

String v/s StringBuffer v/s 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 and keep on changing but thread safety is not required then we should go for StringBuilder.

Method chaining:

For most of the method in String, StringBuffer, 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 form left to right

```
StringBuffer sb = new StringBuffer();
sb.append("durga").append("software").append("solution").insert(2,"xyz").reverse().delete(2,10);
System.out.println(sb);
```

Wrapper classes

The main objectives of wrapper classes are

1. To wrap primitive in to object form so that we can handle primitives also just like objects
2. To define several utilities methods which are required for the primitives.

Constructors:

1. Almost all wrapper classes contain two constructors one can take corresponding primitive as argument and other can take String as argument.

Example:

```
Integer I = new Integer(10);
Integer I = new Integer("10");
```

Example:

```
Double d = new Double(10.5);
Double d = new Double("10.5");
```

2. If String argument not representing a number then we will get runtime exception saying number format exception.

Example:

```
Integer I = new Integer("ten");
RE: NumberFormatException
```

3. Float contain 3 constructors with float, double and string arguments

Example:

```
Floating f = new Floating(10.5f);
Floating f = new Floating(:10.f');
Floating f = new Floating(10.5)
Floating f = new Floating("10.5");
```

4. Character class contain only one constructor which can take char argument.

Example:

```
Character ch = new Character('a'); → valid
Character ch = new Character("a"); → invalid
```

5. Boolean class contains two 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.

Example:

```
Boolean b = new Boolean(true);
Boolean b = new Boolean(false);
Boolean b = new Boolean(True); → invalid
Boolean b = new Boolean(durga); → invalid
```

If you are passing string type as argument then case and content both are not important. If the content is case insensitive is Sting of “true” the it is treated as true other wise it is treated as false

Example:

```
Boolean B = new Boolean ("true") → true
Boolean B = new Boolean ("True") → true
Boolean B = new Boolean ("TRUE") → true
Boolean B = new Boolean ("Malaika") → false
Boolean B = new Boolean ("mallika") → false
Boolean B = new Boolean ("jareena") → false
```

```
Boolean x = new Boolean("yes");
Boolean y = new Boolean("no");
System.out.println(x); → false
System.out.println(y); → false
System.out.println(x.equals(y)); → true
```

Wrapper class	Corresponding constructor arguments
Byte	Byte or String
Short	Short or String
Integer	Integer or String
Long	Long or String
**Float	Float or String or Double
Double	Double or String
**Character	Char
**Boolean	Boolean or String

Note:

In all wrapper class `toString()` method overridden to return content directly. In all wrapper classes `.equals()` method overridden for content comparison.

Utility methods:

1. valueOf()
2. xxxValue()
3. parseXxx()
4. toString()

valueOf():

we can use valueOf() methods to create wrapper object for the given primitive or String.

Form 1:

Every wrapper class except character class contains a static value of method to create wrapper object for the given string.

Public static wrapper valueOf(String s);

Example:

```
Integer I = new Integer.valueOf("10");
Double D = new Double.valueOf("10.5");
Boolean B = new Boolean.valueOf("durga");
```

Form 2:

Every integral wrapper class (Byte, Short, Integer, Long) contains the following valueOf() method to create wrapper object for the given specified radix String.

Public static wrapper valueOf(String s, int radix);

The allowed range of radix is 2 to 36.

Example:

```
Integer I = new Integer.valueOf("100",2);
System.out.println(i); → 4
```

```
Integer I = new Integer.valueOf("101",4);
System.out.println(I); → 17
```

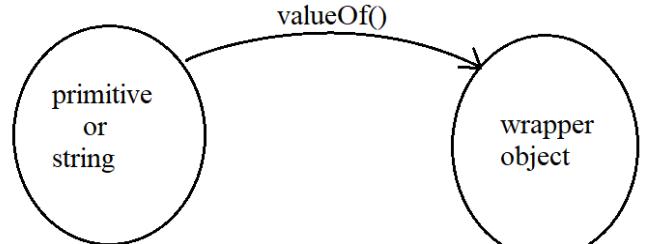
Form 3:

Every wrapper class including Character class contains a static value of method to create wrapper object for the given primitive.

Public static wrapper valueOf(primitive p);

Example:

```
Integer I = Integer.valueOf(10);
Character ch = new Character.valueOf('a');
Boolean B = new Boolean.valueOf(true);
```



xxxValue():

we can use xxxValue() methods to get primitive for the given wrapper object. Every number type wrapper class (byte, short, int, long, float, double) contains the following six methods to get primitive for the given wrapper object.

```
Public byte byteValue()  
Public short shortValue()  
Public int intValue()  
Public long longValue()  
Public float floatValue()  
Public double doubleValue()
```

Example:

```
Integer I = new Integer(130);  
System.out.println(I.byteValue()); → -126  
System.out.println(I.shortValue()); → 130  
System.out.println(I.intValue()); → 130  
System.out.println(I.longValue()); → 130  
System.out.println(I.floatValue()); → 130.00  
System.out.println(I.doubleValue()); → 130.00
```

charValue():

Character class contains charValue() method to get char primitive for the given character object.

```
Public char CharValue()
```

Example:

```
Character c = new Character('a');  
Char ch = c.charValue();  
System.out.println(ch); → a
```

booleanValue():

Boolean class contains booleanValue() method to get Boolean primitive for the given boolean object.

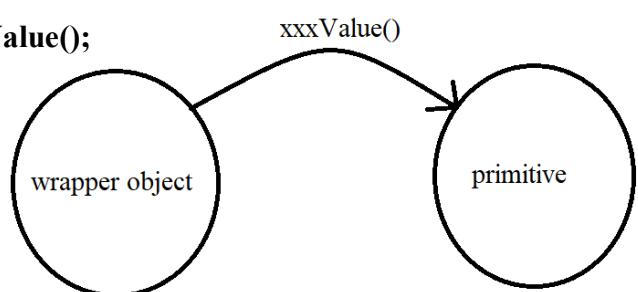
```
Public boolean booleanValue();
```

Example:

```
Boolean B = new Boolean.valueOf("durga");  
Boolean b = B.booleanValue();  
System.out.println(b); → FALSE
```

Note:

In total 38 (= 6 x 6 + 1+1) xxxValue() methods are possible.



parseXxx()

we can use parseXxx() methods to convert string to primitive.

Form 1:

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);

Example:

```
Int I = Integer.parseInt("10");
Double d = Double.parseDouble("10.5");
Boolean b = Boolean.parseBoolean("true");
```

Form 2:

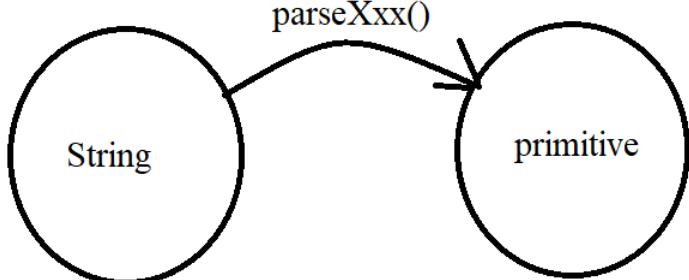
Every integral type wrapper class(byte, short, integer, long) contains the following parseXxx() method to convert specified radix String to primitive.

Public static primitive parseXxx(String s, int radix);

The allowed range of radix is: 2 to 36

Example:

```
Int I = Integer.parseInt("1111",2);
System.out.println(i); // 15
```



toString():

we can use toString() method to convert wrapper object or primitive to String.

Form 1:

Every wrapper class contains the following toString() method to convert wrapper to String type.

Public String toString();

It is the overriding version of object class toString().

When ever trying to print wrapper object reference internally this toString() method will be called

Example:

```
Integer I = new Integer(10);
String s = I.toString();
```

```
System.out.println(s); //10  
System.out.println(l); → System.out.println(l.toString()); →10
```

Form 2:

Every wrapper class including character class contains the following static `toString()` method to convert primitive to string .

```
Public static String toString(primitive p);
```

Example:

```
String s = Integer.toString(10);  
String s = Boolean.toString(true);  
String s = Character.toString('a');
```

Form 3:

Integer and long class 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

Example:

```
String s = Integer.toString(15,2);  
System.out.println(s); //1111
```

Form 4: `toXxxString()`

Integer and Long classes contains the following `toXxxString()` methods.

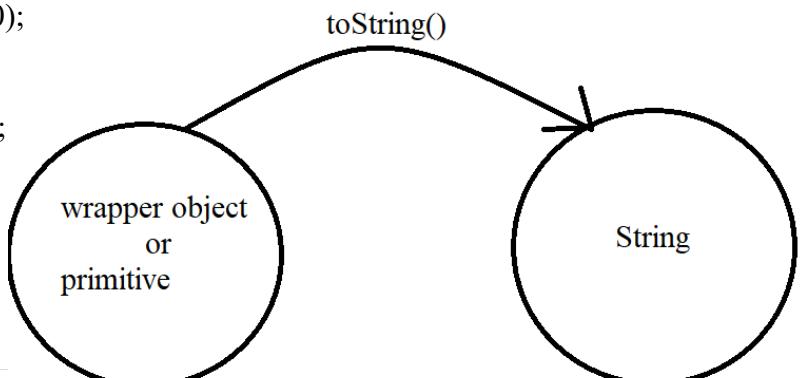
```
Public static String toBinaryString(primitive p);  
Public static String toOctalString(primitive p);  
Public static String toHexString(primitive p);
```

Example:

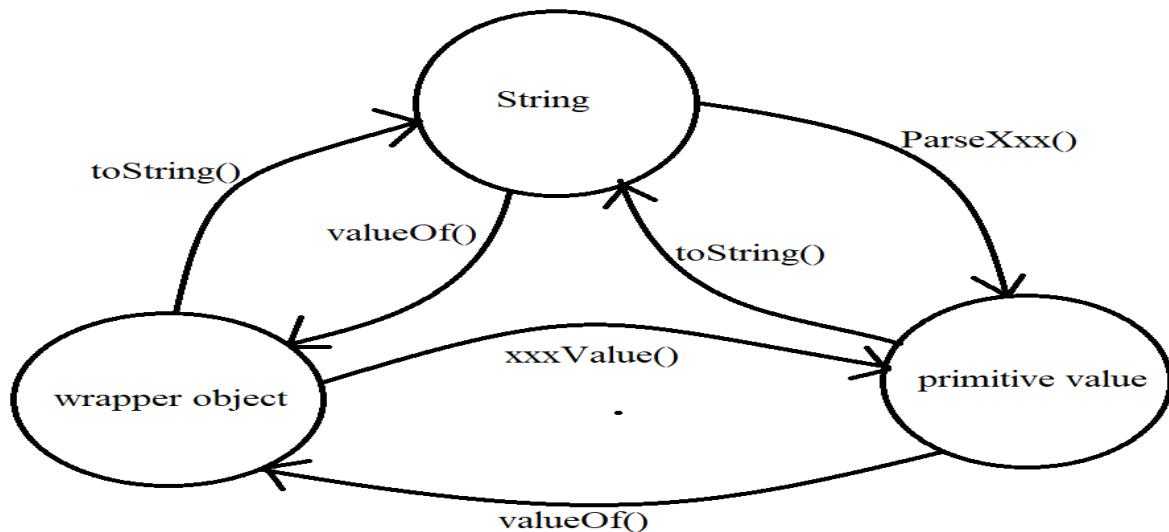
```
String s = Integer.toBinaryString(10);  
System.out.println(s); //1010
```

```
String s = Integer.toOctalString(10);  
System.out.println(s); //12
```

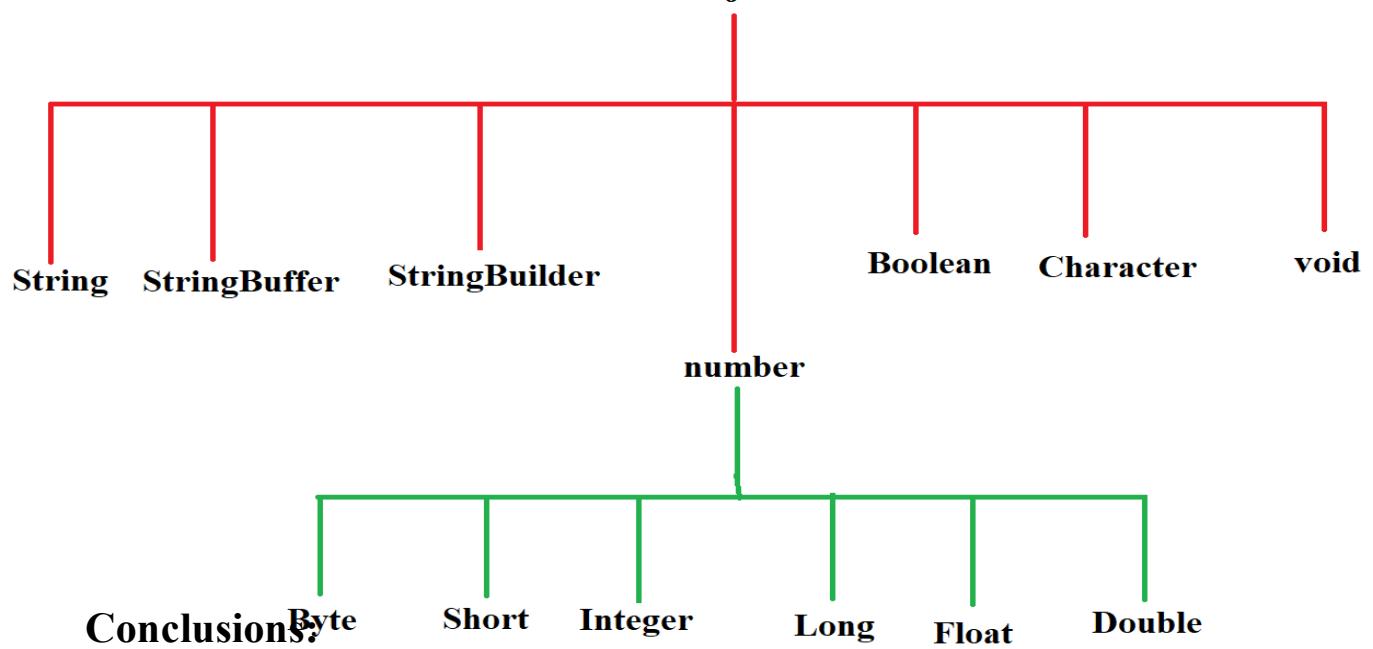
```
String s = Integer.toHexString(10);  
System.out.println(s); //a
```



Dancing between String, wrapper object and primitive:



Partial Hierarchy of `java.lang`. package



- Conclusions:**
- the wrapper classes which are not child class of number are Boolean And Character
 - the wrapper classes which are not direct child class of object are Byte, Short, Integer, Long, Float, Double
 - String, StringBuffer, StringBuilder and all wrapper classes are final classes.
 - In addition to String objects all wrapper class objects also immutable.
 - 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 reflection to check whether the method return type is void or not.

Example:

```
If(getmethod("m1").getReturnType() == Void.type)
{
}
```

Void is the class representation of void keyword in java.

Autoboxing

Automatic conversion of primitive to wrapper object by compiler is called autoboxing.

Example:

```
Integer I = 10; (compiler converts int to Integer automatically by autoboxing)
```

After compilation the above line will be come

```
Integer I = Integer.valueOf(10);
```

That is internally autoboxing concept is implemented by using valueOf() methods

Autounboxing

Automatic conversion of wrapper object to primitive by compiler is called autounboxing.

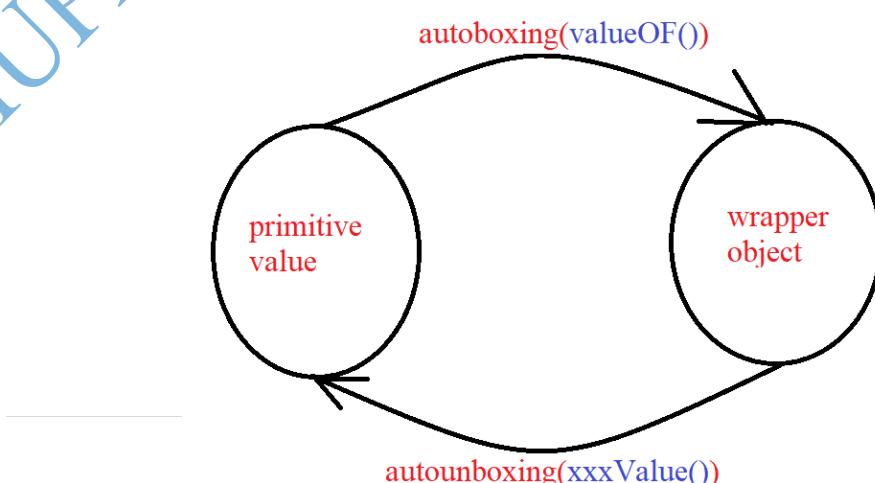
Example:

```
Integer I = new Integer(10);
int i = I; (compiler converts Integer to int automatically by autounboxing)
```

after compilation the above line will become

```
int i = I.intValue();
```

that is internally autounboxing concept is implemented by using "xxxValue()" methods



Example:

```
class AutoBoxing_UnBoxingDemo
{
    static Integer I=10;                                // autoboxing
    public static void main(String[] args)
    {
        int i = I;                                     //autounboxing
        m1(i);
    }
    public static void m1(Integer k)                   //autoboxing
    {
        int m = k;                                     //autounboxing
        System.out.println(m);
    }
} output: 10
```

It is valid in 1.5 version but invalid in 1.4 version.

Note:

Just because of autoboxing and auto-unboxing we can use primitives and wrapper objects interchangeably from 1.5 version onwards

Example 2:

```
class AutoBoxing_UnboxingDemo2
{
    static Integer I=0;
    public static void main(String[]
args)
    {
        int m = I;
        System.out.println(m);
    }
} //output: 0
```

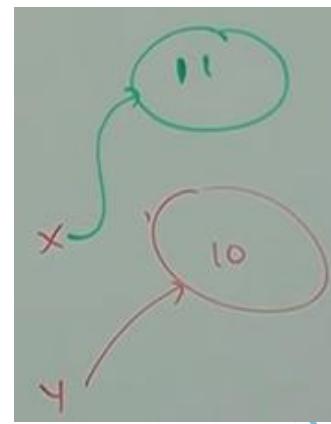
```
class AutoBoxing_UnboxingDemo2
{
    static Integer I;
    public static void main(String[]
args)
    {
        int m = I;
        System.out.println(m);
    }
} //output: RE:
java.lang.NullPointerException
```

Note:

On null reference if we are trying to perform auto-unboxing then we will get runtime exception saying “NullPointerException”.

Example 3:

```
class AutoBoxing_UnboxingDemo3
{
    public static void main(String[] args)
    {
        Integer x=10;
        Integer y =x;
        x++;
        System.out.println(x);      //11
        System.out.println(y);      //10
        System.out.println(x==y);   //false
    }
}
```

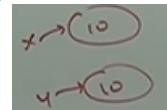


Note:

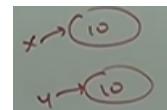
All wrapper class objects are immutable that is once we creates wrapper class object we can't perform any changes in that object. If we are trying to perform any changes with those changes any new object will be created.

Example 4:

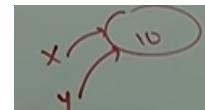
```
class AutoBoxing_UnboxingDemo4
{
    public static void main(String[] args)
    {
        Integer x = new Integer(10);
        Integer y = new Integer(10);
        System.out.println(x==y);//false
    }
}
```



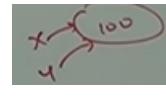
```
Integer x1=new Integer(10);
Integer y1 =10;
System.out.println(x1==y1);//false
```



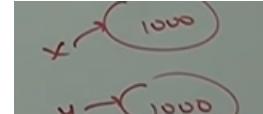
```
Integer x2= 10;
Integer y2= 10;
System.out.println(x2==y2);//true
```



```
Integer x3= 100;
Integer y3= 100;
System.out.println(x3==y3);//true
```



```
Integer x4 =1000;
Integer y4 = 1000;
System.out.println(x4==y4);//false }
```

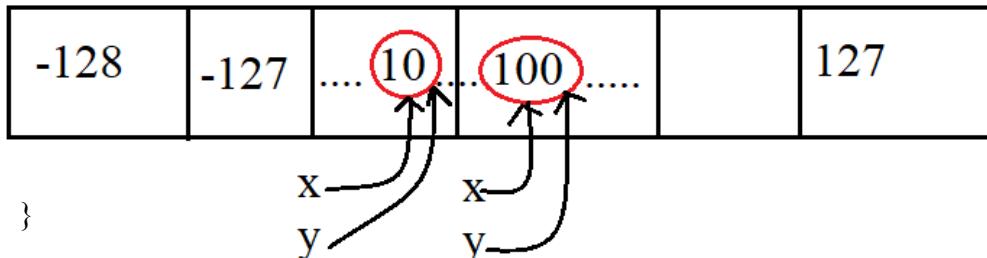


Conclusion:

Internally to provide support for autoboxing a buffer of wrapper objects will be created at the time of wrapper class loading. By auto-boxing if an object is required to create first JVM will check whether this object is present or not, if it is already present in the buffer

then existing buffer object will be used. If it is not already available then JVM will create a new object.

```
Class Integer
{
    Static
    {
        }
```



But buffer concept is available only in the following ranges.

Byte	→	always
Short	→	-128 to +128
Integer	→	-128 to +128
Long	→	-128 to +128
Character	→	0 to 127
Boolean	→	always

Except this range in all remaining cases a new object will be created.

```
Integer x= 127;
Integer y = 127;
System.out.println(x==y);//true

Integer x1 = 128;
Integer y1 = 128;
System.out.println(x1==y1);//false

Boolean x2 = false;
Boolean y2 = false;
System.out.println(x2==y2);//true

Double x3 = 10.0;
Double y3 = 10.0;
System.out.println(x3==y3);//false
```

Internally autoboxing concept is implemented by using valueOf() methods hence buffer concept is applicable for value of methods also

```
Integer x = new Integer(10);
Integer y = new Integer(10);
```

```
System.out.println(x==y);//false
```

```
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
```

```
Integer x = 10;  
Integer y = Integer.valueOf(10);  
System.out.println(x==y);//true
```

Overloading with respect to autoboxing, widening & var-arg methods:

Case 1: autoboxing v/s widening

```
class AutoBoxing_Widening  
{  
    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);  
    }  
}  
//OUTPUT: widening
```

Widening dominates autoboxing.

Case 2: widening v/s var-arg method

```
class Widening_vararg {  
    public static void m1(int... x){  
        System.out.println("var-arg method");  
    }  
    public static void m1(long l){  
        System.out.println("widening one");  
    }  
}
```

```

public static void main(String[] args) {
    int x = 10;
    m1(x);
}
//OUTPUT: widening

```

Widening dominates var- arg methods

Case 3: autoboxing v/s var-arg() methods

```

class Autoboxing_vararg
{
    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);
    }
}
//OUTPUT: autoBoxing

```

Autoboxing dominates var- arg methods

In general, var- arg method will get least priority. That is, if no other method matched then only var -arg method will get the chance it is exactly same as default case in side switch.

Note:

*** while resolving overloaded method compiler will always gives the precedence in the following order

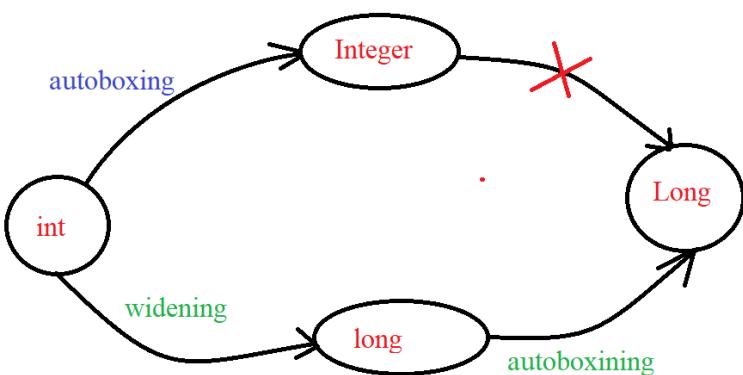
1. Widening
2. Autoboxing
3. Var – arg methods

Case 4:

```

class Var_Weid_AutoDemo
{
    public static void m1(Long l)
    {
        System.out.println("long");
    }
}

```



$W \implies A \implies \text{invalid}$
 $A \implies W \implies \text{valid}$

```

public static void main(String[] args)
{
    int x = 10;
    m1(x);
}
CE: m1(java.lang.Long) in test cannot be applied to (int);

```

Widening followed by autoboxing is not allowed in java whereas autoboxing followed by widening is allowed.

Example:

```

Long l= 10; CE: incompatible types found: int , required: java.lang.Long
Long l=10; → valid

```

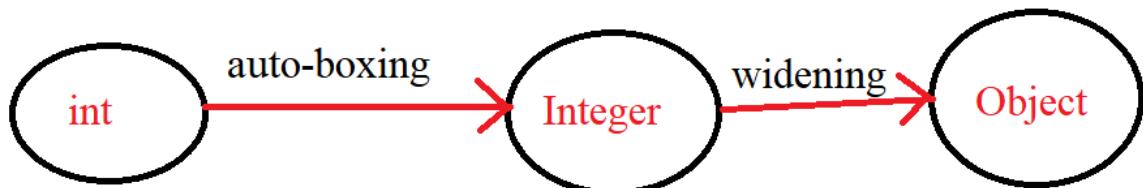
Case 5:

```

class Var_Weid_AutoDemo1
{
    public static void m1(Object o)
    {
        System.out.println("Object version");
    }
    public static void main(String[] args)
    {
        int x = 10;
        m1(x);
    }
}

```

Output:
Object version
Object 0 =10;
Number n = 10; valid in both cases



Which of the following assignments are legal?

- | | |
|----------------|--|
| int i=10; | → valid |
| Integer I= 10; | → valid(autoboxing) |
| Int i = 10L; | → invalid(CE: possible loss of precision found: long required: int); |
| Long l = 10L; | → valid(autoboxing) |
| Long l=10; | → invalid(CE: incompatible types found: int required: Long) |
| long l = 10; | → valid(widening) |
| Object o =10; | → valid(autoboxing followed by widening) |
| double d= 10; | → valid(widening) |
| Double D = 10; | → invalid (CE: incompatible types found: int required: Double) |
| Number n = 10; | → valid(autoboxing followed by widening) |

Relation between == operator and .equals() method:

Conclusion 1:

If two objects are equal by == operator then these objects are always equal by .equals() method. That is

If $r1 == r2$ is true the $r1.equals(r2)$ is always true.

Conclusion 2:

If two objects are not equal by == operator then we can't conclude any thing about .equals() method it may returns true or false that is

If $r1 == r2$ is false then $r1.equals(r2)$ may returns true or false and we can't except exactly

Conclusion 3:

If two objects are equal by .eqals() method then we can't conclude any thing about == operator it may return true or false that is

If $r1.equals(r2)$ is true then $r1 == r2$ is true then we can't conclude any thing about $r1 == r2$ it may returns true false

Conclusion 4:

If two objects are not equal by .equals() method then these objects always not equal by == operator.

If $r1.equals(r2)$ is false then $r1 == r2$ is always false

Difference between == operator and .equals() method:

To use == operator compulsory there should be some relation between argument types (either child to parent or parent to child or same type) other wise we will get compile time error saying "incomparable types"

If there is no relation between argument types then .equals() method won't rise any compile time or run time errors simply it returns false.

Example:

```
class DuffString
{
    public static void main(String[] args)
    {
        String s1 = new String("malachi");
        String s2 = new String("malachi");
        StringBuffer sb1 = new StringBuffer("durga");
        StringBuffer sb2 = new StringBuffer("durga");
        System.out.println(s1==s2);//false
```

```

        System.out.println(s1.equals(s2));//true
        System.out.println(sb1 == sb2);//false
        System.out.println(sb1.equals(sb2));//false
        System.out.println(s1 == sb1);
                                //CE:incomparable types String and StringBuffer
        System.out.println(s1.equals(sb2));//false
    }
}

```

== operator	.equals() method
1. It is an operator in java applicable for both primitives and object types.	1. It is a method applicable only for object types but not for primitives.
2. In the case of object reference == operator meant for reference comparison (address comparison)	2. By default .equals() method present in object class also meant for reference comparison
3. We can't override == equal operator for content comparison.	3. We can override .equals() method for content comparison.
4. To use == operator compulsory there should be some relation between argument type (either child to parent or parent to child or same type) other wise we will get compile time error saying "incomparable types"	4. If there is no relation between argument types then ,equals() method won't raise any compile time or runtime errors and simply returns false.

Answer in one line for interview

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

Note:

For any object reference R, R == null; or r.equals(null) always returns false

Example:

```

Thread t = new Thread();
System.out.println(t == null);//false
System.out.println(t.equals(null));//false

```

Note:

Hashing related data structures follow the following fundamental rule.

To equivalent objects should be placed in same bucket but all objects present in the same bucket need not be equal.

Contract between .equals() method and hashCode() method

1. If two objects are equal by .equals() method then there hash Codes must be equal that is two equivalent object should have same hash code that is

If r1.equals(r2) is true then r1.hashCode() == r2.hashCode() is always true
2. Object class .equals() method and hashCode() method follows above contract hence when ever we are over riding .equals() method compulsory we should override hashCode() method to satisfy above contract(that is two equivalent object should have same hashCode())
3. If two objects are not equal by .equals() method then there is no restriction on hash codes may be equal or may not be equal.
4. If hash code of two objects are equal then we can't conclude any thing about .equals() method it may returns true or false.
5. If hash code of two objects are not equal then these objects are always not equal by .equals() method.

Note:

*** to satisfy contract between equals and hash code method when ever we are overriding .equals() method compulsory we have to override hashCode() method other wise we won't get any compile time 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 hash code based on content

```
class EqualHashCode
{
    public static void main(String[] args)
    {
        String s1 = new String("durga");
        String s2 = new String("durga");
        System.out.println(s1.equals(s2));//true
        System.out.println(s1.hashCode());//95950491
        System.out.println(s2.hashCode());//95950491
    }
}
```

In StringBuffer .equals() method is not overridden for content comparison and hence hashCode() method is also not overridden.

```
class EqualHashCodeStringBuffer
{
    public static void main(String[] args)
    {
        StringBuffer sb1 = new StringBuffer("durga");
        StringBuffer sb2 = new StringBuffer("durga");
        System.out.println(sb1.equals(sb2));//false
    }
}
```

```

        System.out.println(sb1.hashCode());//366712642
        System.out.println(sb2.hashCode());//1829164700
    }
}

```

Consider the following Person class:

```

class Person
{
    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;
            }
        }
        return false;
    }
}

```

Which of the following hash cod methods are appropriate for person class

```

public int hashCode()
{
    return 100;
} → invalid

```

```

public int hashCode()
{
    return age+ssno;
} → invalid

```

```

public int hashCode()
{
    return name.hashCode()+age;
} → valid

```

- No restricted
- In valid

Base on which parameters we override .equals method, it is highly recommended to use same parameter while overriding hashCode() method also

Note:

In 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

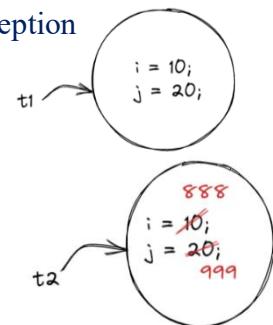
clone():

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

```
protectd native Object clone() throws CloneNotSupportedException
```

```
class Test implements Cloneable
{
    int i = 10;
    nt j = 20;
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Test t1 = new Test();
        Test t2 = (Test)t1.clone();
        t2.i = 888;
        t2.j = 999;
        System.out.println(t1.i+"..."+t1.j);
    }
}
```



We can perform cloning only for cloneable objects.

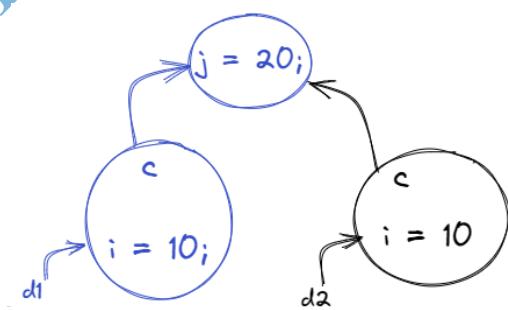
An object is said to be cloneable if and only if the corresponding class implements cloneable interface.

Cloneable interface present in java.lang.package and it doesn't contains any methods it is a marker interface.

If we are trying to perform cloning for non-cloneable object then we will get run time exception saying "CloneNotSupportedException".

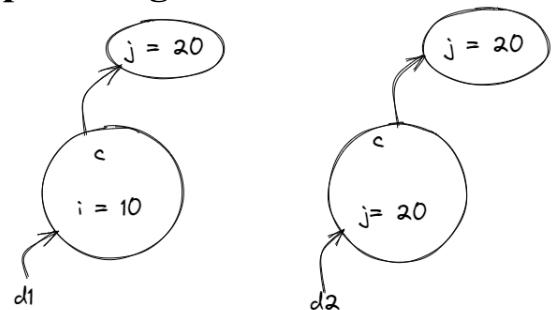
Shallow cloning v/s deep cloning

Shallow cloning



Dog d2= (Dog)d1.clone();

Deep cloning



Dog d2 = (Dog)d1.clone();

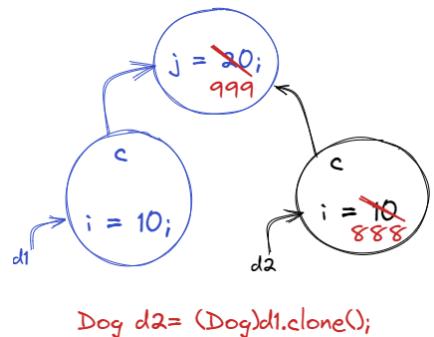
Shallow cloning

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 contains any reference variable, then corresponding object won't be created just duplicate reference will be created pointing to old content object.

Object class clone meant for shallow cloning

```
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 Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}

class ShallowCloningDemo
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Cat c = new Cat(20);
        Dog d1 = new Dog(c,10);
        System.out.println(d1.i+"...."+d1.c.j); //10....20
        Dog d2 = (Dog)d1.clone();
        d2.i=888;
        d2.c.j = 999;
        System.out.println(d1.i+"...."+d1.c.j); //10....999
    }
}
```



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

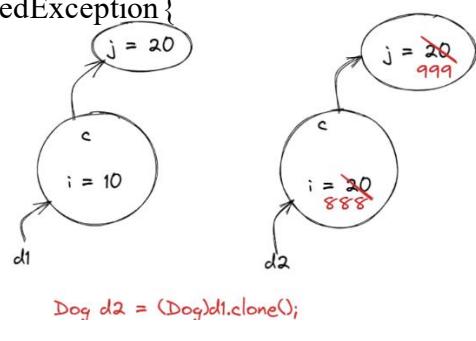
The process of creating exactly duplicate independent copy including contented object is called deep cloning.

In deep cloning if the main object contains any primitive variables, then in the cloned object duplicate copies will be created

If the main object contains any reference variable, then the corresponding contented object also will be created in the cloned copy.

By default object class clone method meant for shallow cloning but we can implement deep cloning explicitly by overriding clone() method in our class.

```
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 Object clone() throws CloneNotSupportedException
    {
        Cat c1 = new Cat(c.j);
        Dog d2 = new Dog(c1,i);
        return d2;
    }
}
class DeepCloningDemo
{
    public static void main(String[] args) throws CloneNotSupportedException {
        Cat c = new Cat(20);
        Dog d1 = new Dog(c,10);
        System.out.println(d1.i+"...."+d1.c.j);//10 .... 20
        Dog d2 = (Dog)d1.clone();
        d2.i=888;
        d2.c.j = 999;
        System.out.println(d1.i+"...."+d1.c.j);// 10 .... 20
        //System.out.println(d2.i+"...."+d2.c.j);//888 .... 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.

```
class StringClassDemo
{
    public static void main(String[] args)
    {
        String s1 = new String("your cannot change me!");
        String s2 = new String("your cannot change me!");
        System.out.println(s1==s2);//false

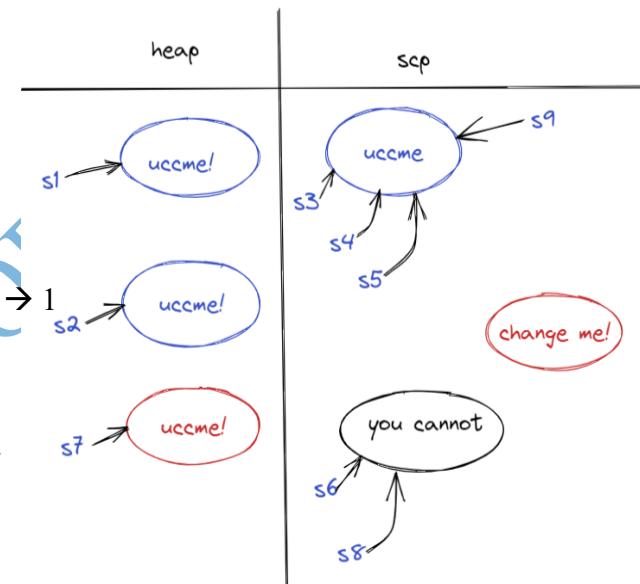
        String s3 = "your cannot change me!";
        System.out.println(s1==s3);//false

        String s4 = "your cannot change me!";
        System.out.println(s3==s4);//true

        String s5 = "your cannot"+"change me!"; → 1
        System.out.println(s3==s5);//true

        String s6 = "your cannot";
        String s7= s6+"change me!";
        System.out.println(s3==s7);//false

        final String s8 = "you cannot";
        String s9 = s8+"change me!";
        System.out.println(s3==s9);//true
        System.out.println(s6 == s8);//true
    }
}
```



At line 1: this operation will be performed at compile time only because both arguments are compile time constants.

At line 2: this operation will be performed at runtime only because at least one argument is normal variable.

At line 3: this operation will be performed at compile time only because both arguments are compile time constants.

Interning of String objects

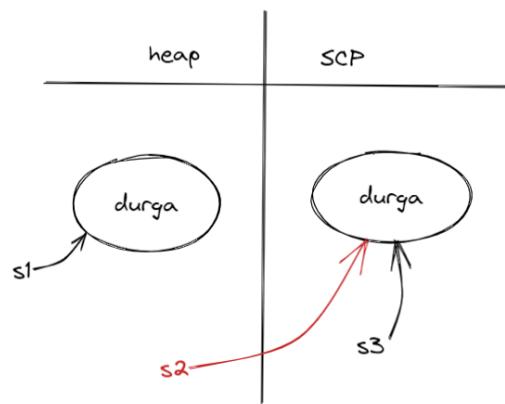
We can use intern() method to get corresponding SCP object reference by using heap object reference

Or

By using heap object reference if we want to get corresponding SCP object reference then we should go for intern() method.

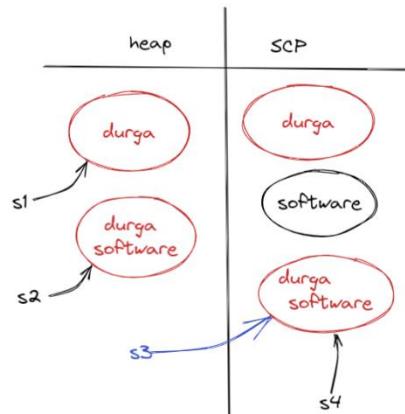
Example:

```
class InterningDemo
{
    public static void main(String[] args)
    {
        String s1= new String("durga");
        String s2 = s1.intern();
        System.out.println(s1==s2);//false
        String s3="durga";
        System.out.println(s2==s3);//true
    }
}
```

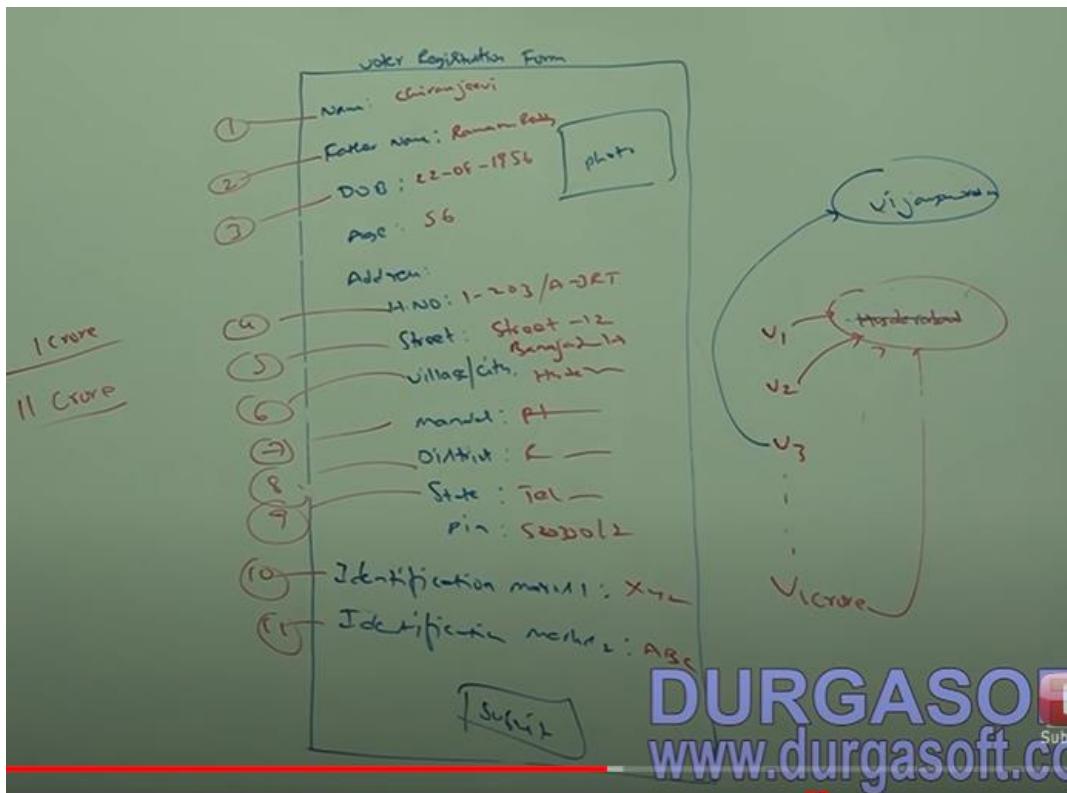


If the corresponding SCP object is not available then intern() method it self create the corresponding SCP object.

```
class InterningDemo1
{
    public static void main(String[] args)
    {
        String s1= new String("durga");
        String s2 = s1.concat("software");
        String s3 = s2.intern();
        System.out.println(s2==s3);//false
        String s4= "durga software";
        System.out.println(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 a 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 utilization will be improved.

This thing is possible because of SCP. Hence the main advantages of SCP are memory utilization and performance will be improved.

But the main problem with SCP is, as several reference pointing to the same object, by using one reference if we are trying to change the content then remaining reference will be affected.

To over come this problem SUN people implemented String objects as immutable that is once we create a String object, we can't perform any changes in the existing object. If we are trying perform changes with those changes a new object will be created. Hence SCP is the only reason for immutability of String objects.

FAQ for interview:

1. What is the difference between String and StringBuffer?
2. Explain about Immutability and mutability with an example?
3. What is the difference between
`String s = new String("durga");` and
`String s = "durga";`
4. Other than immutability and mutability is any other difference between String and StringBuffer?

5. What is SCP?

Ans: It is a special designed memory area for String objects

6. What is the advantage of SCP?

7. What is the disadvantage of SCP?

8. Why SCP like concept is available only for String but not for StringBuffer?

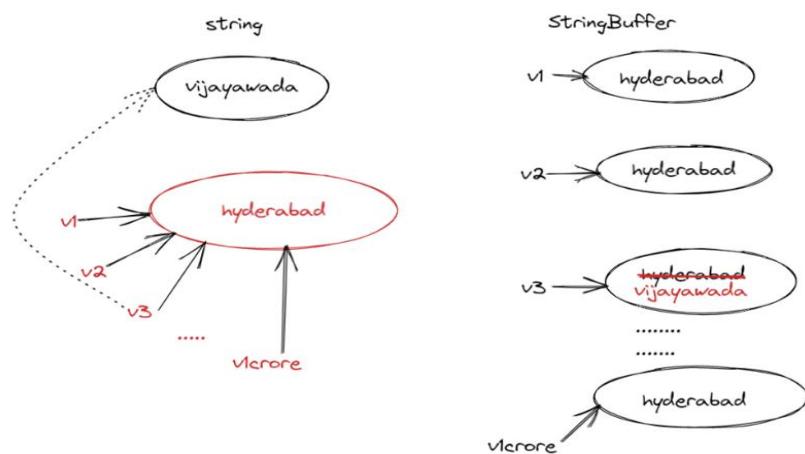
Ans: String is the most commonly used object and hence SUN people provide special memory management for string object

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 mutable?

Ans: 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 reference will be effected to over come this problem sun people implemented String objects as immutable according to this once we creates a string object we can't perform any changes in the existing object if we are trying to perform any change with those changes any new object will be created.



But in String buffer 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 reference hence immutability concept not required for the StringBuffer.

10. In addition to String Objects any other objects are immutable in java?

Ans: in addition to String objects all wrapper class objects also immutable in java

11. Is it possible to create our own immutable class?

Ans: yes

12. How to create our own immutable class? Explain with an example?

13. Immutable means non-changeable where as final means also non-changeable. Then what is the difference between final and immutable?

File I/O

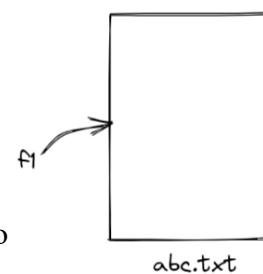
1. File
2. FileWriter
3. File Reader
4. BufferedWriter
5. BufferedReader
6. PrintwWriter

File:

File f new = File("abc.txt");

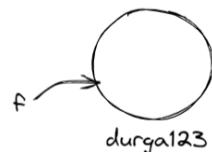
This line won't create an physical file. First it will check is there any physical file named with "abc.txt" is available or not. If It is available then 'f' simply referees that file if it is not available then we are just creating java file object to represent the name "abc.txt"

```
File f = new File("abc123.txt");
System.out.println(f.exists()); //false
f.createNewFile();
System.out.println(f.exists()); //true
```



We can use java file object to represent directory also

```
File f = new File("durga123");
System.out.println(f.exists()); //false
f.mkdir();
System.out.println(f.exists()); // true
```



Note:

In unix every thing is treated as file java file IO concept is implemented based on unix operating system hence java file object can be used to represent both files and directories

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);

Example 1:

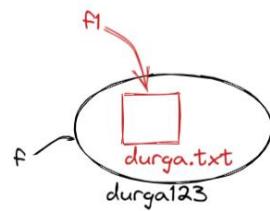
Write code to create a file name with abc.txt in current working directory

```
File f = new File("abc.txt");
f.createNewFile();
```

example 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 f1 = new File("durga123", "demo.txt");
File f2 = new File(f, "Demo.txt");
f1.createNewFile();
```

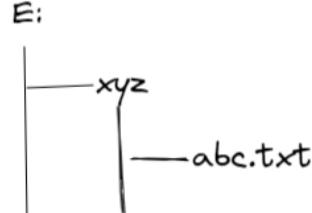


Example 3:

Write code to create a file named with abc.txt in E: xyz Folder

```
File f = new File("E://xyz", "abc.txt");
f.createNewFile();
```

assume E: Xyz folder is available in our system



important method presents in File class:

1. boolean exists():

returns true if the specified file or are 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 mkdir()

4. Boolean isFile():

Return true if the specified file object pointing to physical file

5. Boolean isDirectory()

6. String[] list():

This method returns the names of all files and sub directories present in specified directory

7. Long length()

Return number of characters present in the specified file

8. Boolean delete()

To delete specified file or directory

Write a program to display the names of all files and directory in c:\\ durga _classes

```
import java.io.*;
class ListFiles
{
    public static void main(String[] args) throws Exception
    {
        int count = 0;
        File f = new File("d:\\java programes");
        String[] s = f.list();
        for (String s1 : s)
        {
            count++;
            System.out.println(s1);
        }
        System.out.println("the total number:" + count);
    }
}
```

to display only file names in c:\\ durga _classes

```
import java.io.*;
class OnlyFile
{
    public static void main(String[] args) throws Exception
    {
        int count = 0;
        File f = new File("d:\\java programes");
```

```

String[] s = f.list();
for (String s1:s)
{
    File f1 = new File(f, s1);
    if(f1.isFile()){
        count++;
        System.out.println(s1);
    }
}
System.out.println("the total number:"+count);
}

```

to display only directory names in c:\\ durga _classes

in the above program we have to replace isFile() method with isDirectory() method

import java.io.*;

class OnlyDirectory

{

 public static void main(String[] args) throws Exception

{

 int count = 0;

 File f = new File("d:\\java programs");

 String[] s = f.list();

 for (String s1:s)

{

 File f1 = new File(f, s1);

 if(f1.isDirectory()){

 count++;

 System.out.println(s1);

 }

 }

 System.out.println("the total number:"+count);

 }

}

FileWriter:

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 FileWriters meant for overriding of existing data. Instead of overriding if we want append operation then we have to create file writer by using the following consturctos

3. `FileWriter fw = new FileWriter(String fname, boolean append);`
4. `FileWriter fw = new FileWriter(File f, boolean append);`

Note:

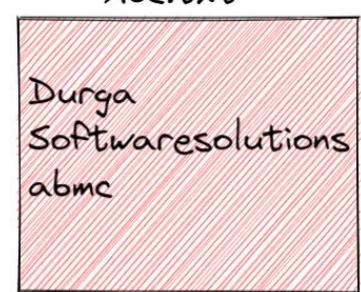
If the specified file is not already available then all the above constructors will create that file.

Methods:

1. **write(int ch)** : To write a single character
2. **write(char[] ch)** : To write an array of characters
3. **write (String s)** : to write String to the file
4. **flush()** : to give the guaranty that total data including last character will be written to file.
5. **Close()** : to close the writer

Example:

```
import java.io.*;
class FileWriterDemo
{
    public static void main(String[] args) throws IOException
    {
        FileWriter fw = new FileWriter("abc.txt");
        fw.write(100); // adding a single character
        fw.write("urga \nsoftware solutions");
        fw.write('\n');
        char[] ch1 = {'a','b','c'};
        fw.write(ch1);
        fw.write('\n');
        fw.flush();
        fw.close();
    }
}
```



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 file writer is we have to insert line separator(\n) manually which is varied from system to system it is defcaulty to the programmer. We can solve this problem by using BufferedWriter and PintWriter classes.

FileReader:

We can use FileReader to read character data from the file

Constructors:

1. FileReader fr = new FileReader(String fname);
2. FileReader fr = new FileReader(File f);

Methods:

1. **Int read();** it attempts to read next character from the file and returns its unique code values. If the next character not available then this method returns -1. As this method returns unique code value(int value), at the time of printing we have perform typecasting.

Example:

```
FileReader fr = new FileReader("abc.txt");
int i = fr.read();
while (i != -1)
{
    System.out.print((char)i);
    i= fr.read();
}
```

2. **Int read(char[] ch);** it attempts to read enough character from the file in to char[] and returns number of characters copied for the file.

Example;

```
import java.io.*;
class FileReaderDemo1
{
    public static void main(String[] args) throws IOException
    {
        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)
        {
            System.out.print(ch1);
        }
    }
}
```

3. Void close();

Example:

```
import java.io.*;
class FileReaderDemo2
{
    public static void main(String[] args) throws IOException
    {
        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)
        {
            System.out.print(ch1);
        }
        System.out.println("*****");
        FileReader fr1 = new FileReader("abc.txt");
        int i = fr1.read();
        while(i!=-1)
        {
            System.out.print((char)i);
            i = fr1.read();
        }
    }
}
```

Note:

By using FileReader we can read data character by character which is not convenient to the programmer.

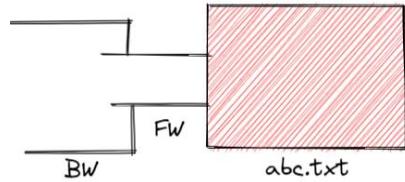
Usage of FileWriter and FileReader is not recommended because

1. While writing data by FileWriter we have to insert line separator (\n) manually which is varied from system to system it is the difficult to the programmer.
2. By using FileReader we can read data character by character, which is not convenient to the programmer.

To over come this problems, we should go for BufferedWriter and BufferedReader.

BufferedWriter:

We can use BufferedWriter to write character data to the file.



Constructors:

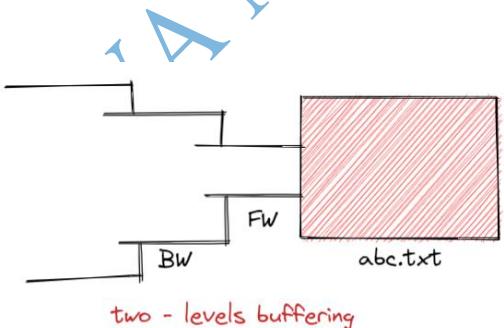
1. BufferedWriter bw = new BufferedWriter(writer w);
2. BufferedWriter bw = new BufferedWriter(writer w, int buffersize)

Note:

BufferedWriter can't communicate directly with the file it can communicate via some writer object.

Q) which of the following are valid

1. BW bw = new BW("abc.txt"); → invalid
2. BW bw = new (new File("abc.txt")); → invalid
3. BW bw = new BW (new FW("abc.txt")); → valid
4. BW bw = new BW (new BW(new FW("abc.txt"))); → valid



Methods:

1. **write(int ch)** : To write a single character
2. **write(char[] ch)** : To write an array of characters
3. **write (String s)** : to write String to the file
4. **flush()** : to give the guaranty that total data including last character will be written to file.
5. **Close()** : to close the writer
6. **newLine()**: to insert a line separator.

Q) when compared with FileWriter which of the following capability available extra in method form in BufferedWriter.

1. writing data to the file
2. close the file
3. flushing the file
4. inserting a new line character → valid

Example:

```
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(100);
```



```

        bw.newLine();
        char[] ch1 = {'a','b','c','d'};
        bw.write(ch1);
        bw.newLine();
        bw.write("durga");
        bw.newLine();
        bw.write("Software Solutions");
        bw.flush();
        bw.close();
    }
}

```

Note:

When ever we are closing BufferedWriter automatically internal FileWriter will be closed and we are not required to close explicitly



BufferedReader:

We can use BufferedReader to read char 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.

Constructor:

1. BufferedReader br = new BufferedReader(Reader r);
2. BufferedReader br = new BufferReader(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():

It attempts to read next line from the file and returns it. If the next line is not available then this method returns null;

Example:

```

import java.io.*;
class BufferedReaderDemo
{
    public static void main(String[] args) throws IOException
    {
        FileReader fr = new FileReader("abc.txt");
        BufferedReader br = new BufferedReader(fr);
        String line = br.readLine();
        while(line != null)
        {
            System.out.println(line);
            line = br.readLine();
        }
        br.close();
    }
}

```

Note:

When ever we are closing BufferedReader automatically underlaying 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.

PrintWriter:

It is the most enhanced writer to write character data to the file. The main advantage of PrintWriter over FileWriter and BufferedWriter we can write any type 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(*inc 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*);

example:

```
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);
        out.println(true);
        out.println('c');
        out.println("durga");
        out.flush();
        out.close();
    }
}
```



Q) What is the difference write(100) & print(100)?

ANS) in the case write(100) the corresponding character 'd' will be added to the file but in case of print(100) the int value '100' will be added to the file directly.

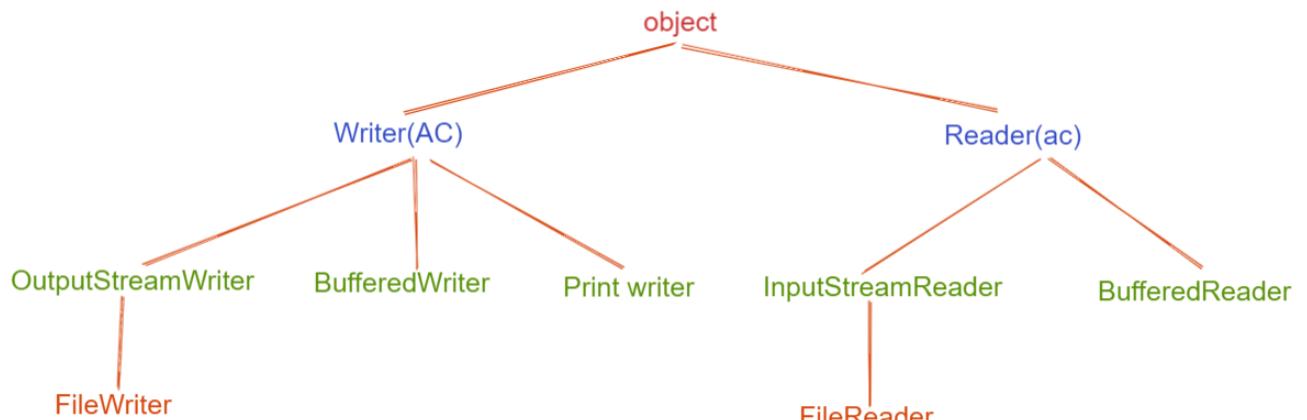
Note:

The most enhanced writer to write character data to the file is PrintWriter where as the most enhanced reader to read character data from the file is BufferedReader.

Note:

in general we can use readers and writers to handle character data(text data), where as we can use streams to handle binary data(like images, pdf, videos... etc)

we can use outputStream to write binary data to the file inputStream to read binary data from the file



Write a program to merge data from two files in to a third file?

```
import java.io.*;
class FileMerger
{
    public static void main(String[] args) throws IOException
    {
        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();
    }
}
```

AAA
BBB
CCC
DDD
EEE
FFF
GGG
HHH

file1.txt

111
222
333
444
555
666
777
888

file2.txt

AAA
BBB
CCC
DDD
EEE
FFF
GGG
HHH
111
222
333
444
555
666
777
888

Write a program to perform file merge operation where merging should be done line by line alternatively.

```
import java.io.*;
class FileMergerLineByLine
{
    public static void main(String[] args) throws IOException
    {
        PrintWriter pw = new PrintWriter("file3.txt");
        BufferedReader br1 = new BufferedReader( new FileReader("file1.txt"));
        BufferedReader br2 = new BufferedReader(new FileReader("file2.txt"));
        String line1 = br1.readLine();
        String line2 = br2.readLine();
        while((line1 != null)|| (line2 != null))
        {
            if (line1 != null)
            {
                pw.println(line1);
                line1 = br1.readLine();
            }
            if(line2 != null)
            {
                pw.println(line2);
                line2 = br2.readLine();
            }
        }
        pw.flush();
        br1.close();
        br2.close();
        pw.close();
    }
}
```

Write a program to perform file extraction operation?

AAA
BBB
CCC
DDD
EEE
FFF
GGG

file1.txt

111
222
333
444
555
666
777
888

file2.txt

AAA
111
BBB
222
CCC
333
DDD
444
EEE
555
FFF
666
GGG
777
888

file3.txt

111
222
333
444
555
666
777

input.txt

222
444
111

delete.txt

333
555
666
777

output.txt

output = input - delete

```

import java.io.*;
class FileExtractor
{
    public static void main(String[] args) throws IOException
    {
        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("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();
        }
        pw.flush();
    }
}

```

Write a java program to remove duplicate from the give input file

```

import java.io.*;
class FileDeleteDuplicateValues
{
    public static void main(String[] args) throws IOException
    {
        PrintWriter pw = new PrintWriter("output1.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("output1.txt"));
            String target = br2.readLine();
            while(target != null)
            {
                if ( line.equals(target))

```

```

        {
            available = true;
            break;
        }
        target = br2.readLine();
    }
    if(available == false)
    {
        pw.println(line);
        pw.flush();
    }
    line = br1.readLine();
}
pw.flush();
}
}

```

111
111
222
222
333
333
444
444

input.txt

111
222
333
444

output.txt

serialization

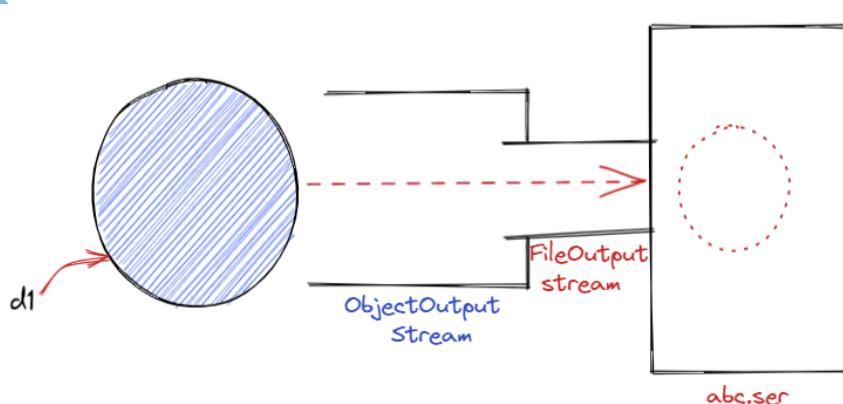
1. introduction
2. object graphs in serialization
3. customized serialization
4. serialization with respect to inheritance
5. externalization
6. serialVersionUID

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.

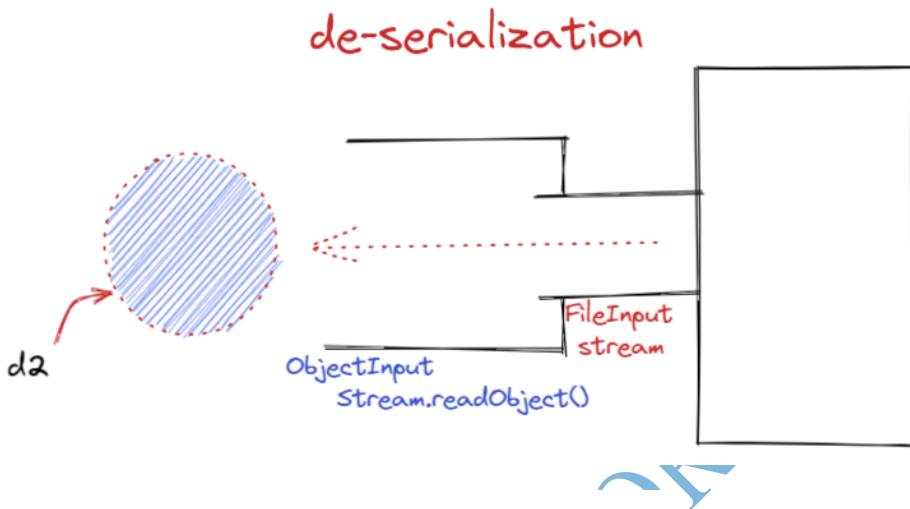
By using FileOutputStream and ObjectOutputStream classes we can implement serialization.



De- serialization

The process of reading state of an object from the file is called de-serialization. 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 de-serialization

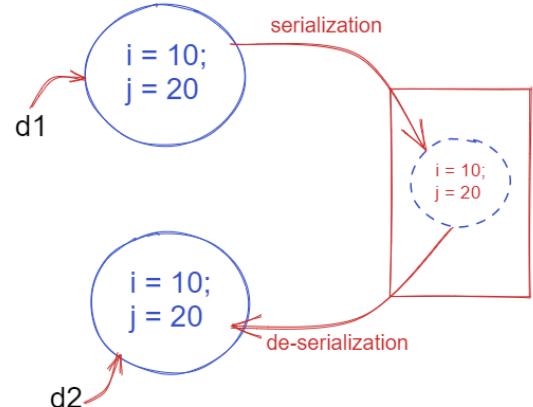


```
import java.io.*;
class Dog implements Serializable
{
    int i = 10;
    int j = 20;
}
class SerializeDemo
{
    public static void main(String[] args) throws Exception
    {
        Dog d1 = new Dog();

        serialization
        FileOutputStream fos = new FileOutputStream("abcd.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        Note: de-serialization
        FileInputStream fis = new FileInputStream("abcd.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();

        System.out.println(d2.i+"...."+d2.j);
    }
}
```



We can serialize only serializable objects. An object is said to be serializable if and only if the corresponding class implements Serializable interface.

Serializable interface present in java.io package and it doesn't contains any methods and it is a marker interface.

If we are trying to serialize a non- serializable object then we will get Runtime exception saying “ NotSerializableException”.

Transient keyword:

Transient modifier 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 we should declare that variable as transient

While performing serialization JVM ignores original value of transient variable and save default value to the file. Hence **transient means not to serialize**

Transient v/s 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 v/s transient

Final variables will be participated in serialization directly by the value hence declaring a final variable as transient there is no impact

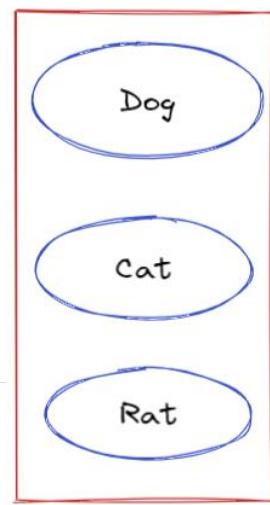
Summary:

Declaration	Output
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 20
Transient int I = 10; Transient final int j = 20	0 20
Transient static int I = 10; Transient final int j = 20	10 20

Note:

We can serialize any number of objects to the file but in which order we serialized in the same order only we have to de-serialize that is order of objects is important in serialization.

```
Dog d1 = new Dog();
Cat c1 = new Cat();
Rat r1 = new Rat();
FileOutputStream fos = new FileOutputStream("abcde.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(d1);
oos.writeObject(c1);
oos.writeObject(r1);
FileInputStream fis = new FileInputStream("abcde.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 object in serialization

```

FileInputStream fis = new FileInputStream("abcde.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 graphs in serialization

When ever we are serializing 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.

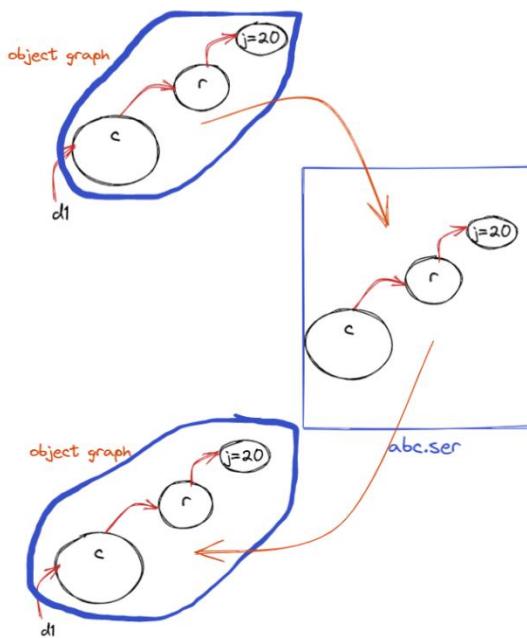
In object graph every object should be serializable if at least one object is not serializable the we will get runtime exception saying "NotSerializableException".

Example:

```

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("abcde.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        FileInputStream fis = new FileInputStream("abcde.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();

        System.out.println(d2.c.r.j);
    }
}

```

In the above program when ever we are serializing Dog object automatically Cat and Rat object got serialized because these are part of object graph of Dog

Among Dog, Cat, Rat objects if at lease one object is not serializable then we will get runtime exception saying “NotSerializableException”

Customized serialization

During default serialization there may be a chance of loss of information because of transient keyword

```

import java.io.*;
class Account implements Serializable
{
    String userName = "bhupathi";
    transient String pwd = "malachi";
}
class CustomizeSerializeDemo
{
    public static void main(String[] args) throws Exception
    {
        Account a1 = new Account();
        System.out.println(a1.userName+"....."+a1.pwd);
        FileOutputStream fos = new FileOutputStream("abcdef.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);

        FileInputStream fis = new FileInputStream("abcdef.ser");

```

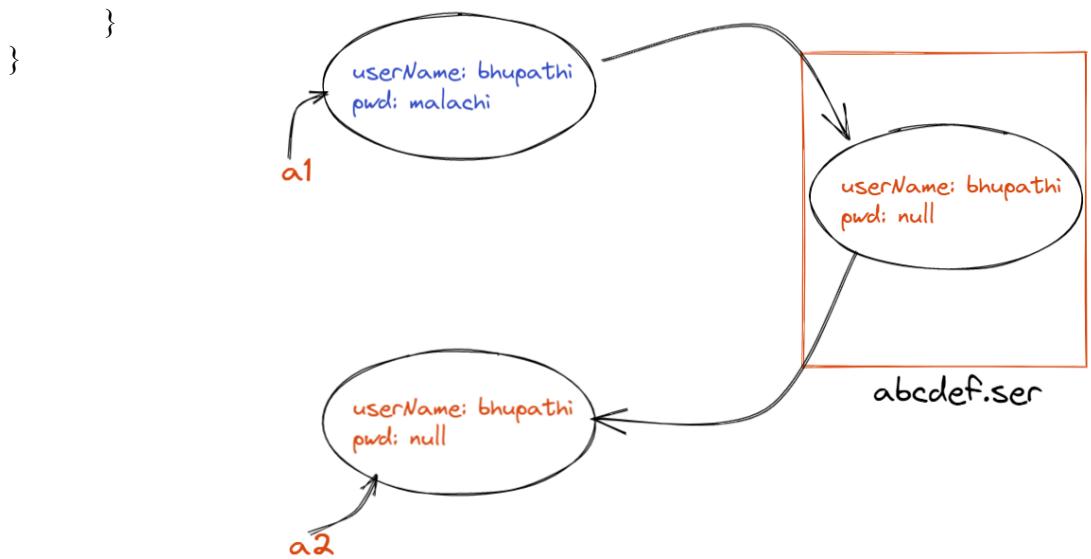
```

ObjectInputStream ois = new ObjectInputStream(fis);
Account a2 = (Account)ois.readObject();

System.out.println(a2.userName+"...."+a2.pwd);

}

```



In the above example before serialization account object can provide proper username and password after deserialization account object can provide only username but not password this is due to declaring password variable as transient

Hence during default serialization there may be a chance of loss of information because of transient keyword to recover the loss of information we should go for customized serialization

We can implement customized serialization the following two methods.

1. Private void writeObject(ObjectOutputStream os) throws Exception

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 de-serialization hence at the time of de-serialization if we want to perform any activity we have to define that in this method only

Note:

The above methods are call back methods because these are executed automatically by the JVM.

While performing which object serialization we have to do extra work in the corresponding class we have define above methods
 for example, while performing account object serialization if we required to do extra work in the account class we have define above methods

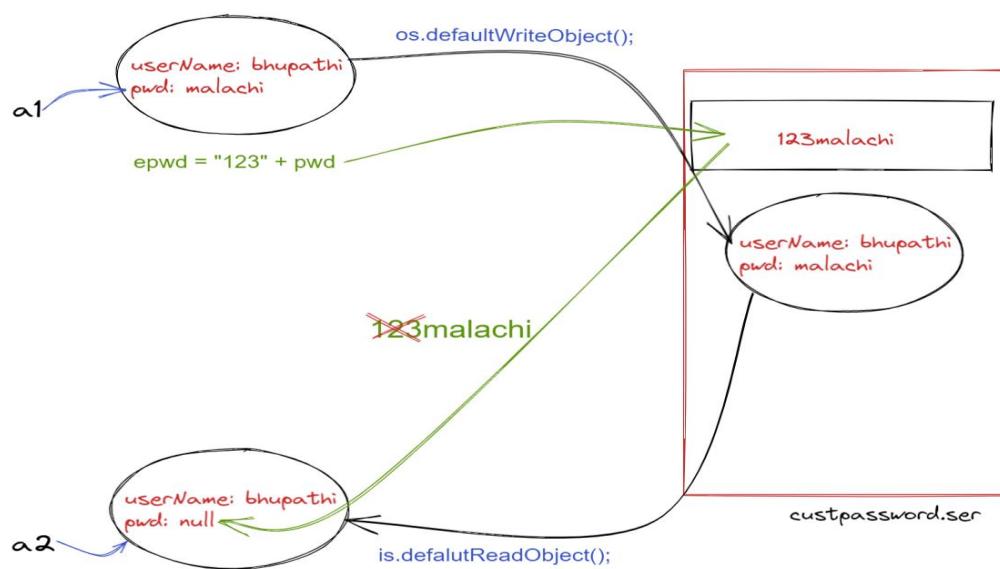
```

import java.io.*;
class Account implements Serializable{
    String userName = "bhupathi";
    transient String pwd = "malachi";
    private void writeObject(ObjectOutputStream os) throws Exception {
        os.defaultWriteObject();
        String epwd = "123"+pwd;
        os.writeObject(epwd);
    }
    private void readObject(ObjectInputStream is) throws Exception {
        is.defaultReadObject();
        String epwd = (String)is.readObject();
        pwd = epwd.substring(3);
    }
}
class CustomizedSerializationDemo1 {
    public static void main(String[] args) throws Exception {
        Account a1 = new Account();
        System.out.println(a1.userName+"...."+a1.pwd);

        FileOutputStream fos = new FileOutputStream("custpassword.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);

        FileInputStream fis = new FileInputStream("custpassword.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Account a2 = (Account)ois.readObject();

        System.out.println(a2.userName+"....."+a2.pwd);
    }
}
    
```



In the above program before serialization after serialization account object can provide proper username and password

Note:

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 2:

```

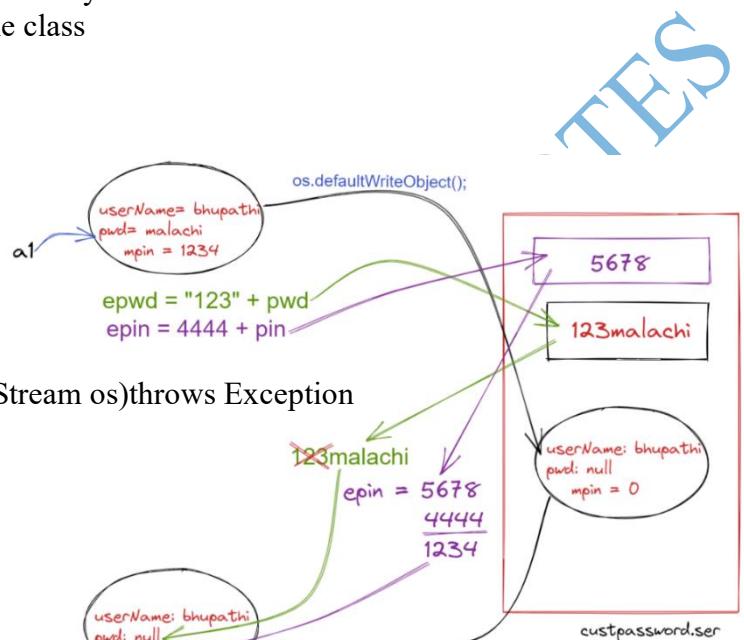
import java.io.*;
class Account implements Serializable
{
    String userName = "bhupathi";
    transient String pwd = "malachi";
    transient int mpin = 1234;
    private void writeObject(ObjectOutputStream os) throws Exception
    {
        os.defaultWriteObject();
        String epwd = "123"+pwd;
        os.writeObject(epwd);
        int epin = 4444 + mpin;
        os.writeInt(epin);
    }
    private void readObject(ObjectInputStream is) throws Exception
    {
        is.defaultReadObject();
        String epwd = (String)is.readObject();
        pwd = epwd.substring(3);
        int epin = is.readInt();
        mpin = epin - 4444;
    }
}
class CustomizedSerializationDemo2
{
    public static void main(String[] args) throws Exception
    {
        Account a1 = new Account();
        System.out.println(a1.userName+"....."+a1.pwd+"...."+a1.mpin);

        FileOutputStream fos = new FileOutputStream("custpassword1.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);

        FileInputStream fis = new FileInputStream("custpassword1.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Account a2 = (Account)ois.readObject();

        System.out.println(a2.userName+"....."+a2.pwd+"...."+a2.mpin);
    }
}

```



Serialization with respect to Inheritance

Case 1:

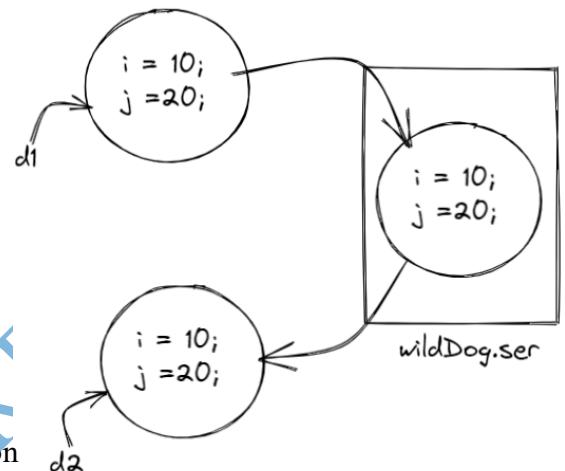
Even though child doesn't implement serializable we can serialize object if parent class implements serializable interface that is serializable nature is inheriting from parent to child hence if parent is serializable then by default every child is serializable.

```
import java.io.*;
class Animal implements Serializable
{
    int i = 10;
}
class WildDog extends Animal
{
    int j = 20;
}
class InheritanceSerialization
{
    public static void main(String[] args) throws Exception
    {
        WildDog d1 = new WildDog();
        System.out.println(d1.i+"....."+d1.j);

        FileOutputStream fos = new FileOutputStream("wildDog.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        FileInputStream fis = new FileInputStream("wildDog.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        WildDog d2 = (WildDog)ois.readObject();

        System.out.println(d2.i+"....."+d2.j);
    }
}
```



In the above example even though WildDog class doesn't implement serializable we can serialize Wild Dog object because it's parent Animal class implements serializable.

Note:

Object class doesn't implement serializable interface

Case 2:

Even though parent class doesn't implement serializable we can serialize child class object if child class implements serializable interface that is to serialize child class object parent class need not be serializable.

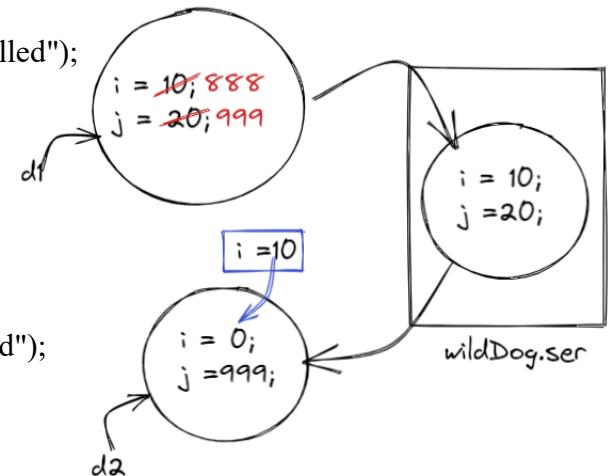
At the time serialization JVM will check is any variable inheriting from non-serializable parent or not if any variable inheriting from non-serializable parent then JVM ignores original value and save default value to the file.

At the time deserialization is any parent class is non-serializable or not if any parent class is non-serializable then JVM will execute instance control flow in every non-serializable parent and share its instance variable to the current object.

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"

```
import java.io.*;
class Animal
{
    int i = 10;
    Animal()
    {
        System.out.println("animal constructor called");
    }
}
class WildDog extends Animal implements Serializable
{
    int j = 20;
    WildDog()
    {
        System.out.println("Dog constructor called");
    }
}
class InheritanceSerialization1
{
    public static void main(String[] args) throws Exception
    {
        WildDog d1 = new WildDog();
        d1.i = 888;
        d1.j = 999;
        System.out.println(d1.i+"...."+d1.j);

        FileOutputStream fos = new FileOutputStream("wildDog.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);
    }
}
```



```

        System.out.println("Deserialization started");

        FileInputStream fis = new FileInputStream("wildDog.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        WildDog d2 = (WildDog)ois.readObject();

        System.out.println(d2.i+"....."+d2.j);
    }
}

```

Output:

```

animal constructor called
Dog constructor called
888.....999
Deserialization started
animal constructor called
10.....999

```

Externalization

In serialization every thing takes care by JVM and programmer doesn't have any control.

In serialization is always possible to save total object to the file and it is not possible to save part of the object, which make creates performance problems.

To over come this problem we should go for externalization.

The main advantage of externalization over serialization is every thing takes care by programmer and JVM doesn't have any control.

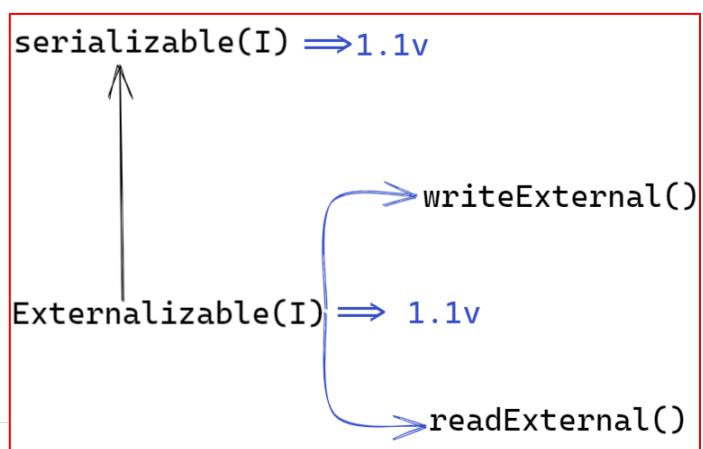
Base on our requirement we can save either total object or part of the object, which improves performance of the system.

To provide externalizable ability for any java object compulsory the corresponding class should implement externalizable interface.

Externalizable interface defines two methods

1. writeExternal()
2. readExternal()

Externalizable is the child interface of serializable.



public void writeExternal(ObjectOutput out) throws IOException

this method will be executed automatically at the time of serialization. Within this method we have to write code to save required variables to the file.

public void readExternal(ObjectInput in) throws IOException

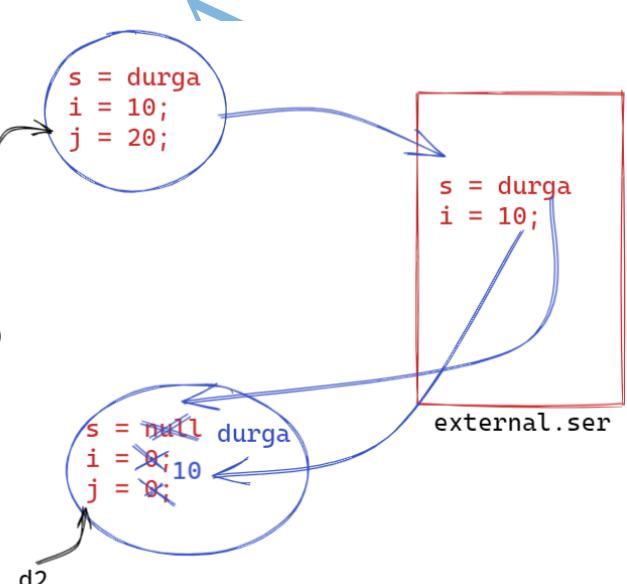
this method will be executed automatically at the time of deserialization. Within this method, we have to write code to read required variable from the file and assign to current object.

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 runtime exception saying "InvalidClassException".

Example:

```
import java.io.*;
class ExternalizableDemo implements Externalizable
{
    String s ;
    int i;
    int j;
    public ExternalizableDemo(String s , int i, int j)
    {
        this.s = s;
        this.i = i;
        this.j = j;
    }
    public ExternalizableDemo()
    {
        System.out.println("public no-argument constructor");
    }
    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 d1 = new ExternalizableDemo("durga",10,20);

        FileOutputStream fos = new FileOutputStream("external.ser");
```



```
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(d1);

InputStream fis = new FileInputStream("external.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
ExternalizableDemo d2 = (ExternalizableDemo)ois.readObject();

System.out.println(d2.s+"...."+d2.i+"...."+d2.j);
}
}
```

Output:

public no-argument constructor
durga.....10.....0

if the class implements serializable then total object will be saved to the file in this case output is

durga 1020

if the class implements externalizable the only required variables will be saved to the file in this case output is

public no-argument constructor
durga 10 0

note:

In serialization transient keyword will play role but in externalization transient keyword won't play any role off course transient keyword not required in externalization

Difference between serialization and externalization

Serialization	Externalization
1. it is meant for default serialization	1. it is meant for customized serialization
2. here every thing takes care by JVM and programmer doesn't have any control.	2. Here every thing 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. Base on our requirement we can save either total object or part of the object.
4. Relatively performance is low	4. Relatively performance is high
5. It is the best choice if we want to save total object to the file.	5. It is the best choice if we want to save part of the object to the file.
6. Serializable interface doesn't contain any methods and it is a marker interface	6. Externalizable contains two methods writeExternal() and readExternal() and hence it is not a marker interface
7. Serializable implemented class not required to contain public no-argument constructor	7. Externalizable implemented class should compulsorily contain public no-argument constructor. otherwise, runtime exception saying "InvalidClassException"
8. Transient keyword will play role in serialization.	8. Transient keyword won't play any role in externalization off course it wont be required.

serialVersionUID

In serialization both sender and receiver need not be same person need not 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 files at the beginning only just state of object is traveling from sender to receiver.

At the time serialization with every object sender side JVM will save a unique identifier. JVM is responsible to generate this unique identifier based on .class file.

At the time deserialization receiver side JVM will compare unique identifier associated with object with local class unique identifier if both are matched then only deserialization will be performed other wise we will get runtime exception saying "InvalidClassException". This unique identifier 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, platform and version otherwise receiver unable to deserialize because of different serial version UID'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 serial version UID internally JVM may use complex algorithm which may create performance problem.

We can solve above problems by configuring our own serialVersionUID

We can configure our serialVersionUID as follows

```
private static final long serialVersionUID = 1L;
```

DogOne.java

```
import java.io.*;  
class DogOne implements Serializable  
{  
    private static final long serialVersionUID =1L;  
    int i = 10;  
    int j = 20;  
}
```

Sender.java

```
import java.io.*;  
class Sender  
{  
    public static void main(String[] args) throws Exception  
    {  
        DogOne d1 = new DogOne();  
        FileOutputStream fos = new FileOutputStream("svuid.ser");  
        ObjectOutputStream oos = new ObjectOutputStream(fos);  
        oos.writeObject(d1);  
    }  
}
```

Receiver.java

```
import java.io.*;
class Receiver
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis = new FileInputStream("svuid.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        DogOne d2 = (DogOne)ois.readObject();
        System.out.println(d2.i+"....."+d2.j);
    }
}
```

In the above program after serialization if we perform any change to the .class file at receiver side we wont get any problem at the time deserialization.

In this case sender and receiver not required to maintain same JVM version.

Note:

Some IDE's prompt programmer to enter serialVersionUID explicitly.

Note:

Some IDE's may generate serialVersionUID automatically

Regular Expression

If we want to represent a group of Strings according to a particular pattern then we should go for Regular Expression.

Example1:

We can write a regular expression to represent all valid mobile numbers

Example 2:

We can write a regular expression to represent all mail id's

The main important application areas of regular expressions are

1. To develop validation frame works
2. To develop pattern matching applications(ctrl + f in windows and grep in Unix)
3. To develop translators like assemblers, compilers, interpreters etc...
4. To develop digital circuits.
5. To develop communication protocol like TCP/IP, UDP etc.

```
import java.util.regex.*;
class RegularExpressionDemo
{
    public static void main(String[] args)
    {
        int count =0;
        Pattern p = Pattern.compile("ab");
        Matcher m = p.matcher("abbabbbabbbba");
        while(m.find())
        {
            count++;
            System.out.println(m.start()+"....."+m.end()+"....."+m.group());
        }
        System.out.println("the total number of occurrences:"+count);
    }
}
```

Output:

```
0.....2.....ab
3.....5.....ab
7.....9.....ab
the total number of occurrences:3
```

Pattern:

A pattern object is a compiled version of regular expression that is 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)

Example;

```
Pattern p = new 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() of pattern class

public Matcher matcher(String target)

example;

```
Matcher m = p.matcher("abbabbba");
```

Important methods of Matcher class:

boolean find()

it attempts to find next match and returns true if it is available

int start()

returns start index of the match

int end()

returns end +1 index of the match

String group()

It returns the matched patterns

NOTE: pattern and matcher classes present in `java.util.regex` package and introduced in 1.4 version.

Character classes:

[abc]	→ either 'a' or 'b' or 'c'
[^abc]	→ except 'a' and 'b' and 'c'
[a-z]	→ any lower case alphabet symbol for a to z
[A-Z]	→ any upper case alphabet symbol from A to Z
[a-zA-Z]	→ any alphabet symbol
[0-9]	→ any digit from 0 to 9
[0-9a-zA-Z]	→ any alphanumeric symbol
[^0-9a-zA-Z]	→ except alphanumeric characters is nothing but special symbols

example

```
import java.util.regex.*;
class RegularExpressionDemo1
{
    public static void main(String[] args)
    {
        Pattern p = Pattern.compile("[0-9a-zA-Z]");
        Matcher m = p.matcher("a3b#k@9z");
        while(m.find())
        {
            System.out.println(m.start()+"....."+m.group());
        }
    }
}
```

Output:

X = [abc]	X= [^abc]	X=[a-z]	X= [0-9]	X=[a-zA-Z]	X= [^a-zA-Z0-9]
0.....a	1.....3	0.....a	1.....3	0.....a	3.....#
2.....b	3.....#	2.....b	6.....9	1.....3	5.....@
	4.....k	4.....k		2.....b	
	5.....@	7.....z		4.....k	
	6.....9			6.....9	
	7.....z			7.....z	

Pre defined character classes:

\s	=> space character
\S	=> Except space character
\d	=> any digit from 0 – 9
\D	=> except digit, any character
\w	=> any word character[0-9a-zA-Z]
.	(dot) => any character

```

import java.util.regex.*;
class RegularExpressionDemo2
{
    public static void main(String[] args)
    {
        Pattern p = Pattern.compile("\s");
        Matcher m = p.matcher("a7b k@9z");
        while(m.find())
        {
            System.out.println(m.start()+"....."+m.group());
        }
    }
}

```

Output:

$X=\backslash s$	$X= \underline{\backslash S}$	$X=\backslash d$	$X=\backslash D$	$X=\backslash w$	$X=.$
3.....	0.....a	1.....7	0.....a	0.....a	0.....a
	1.....7	6.....9	2.....b	1.....7	1.....7
	2.....b		3.....	2.....b	2.....b
	4.....k		4.....k	4.....k	3.....
	5.....@		5.....@	6.....9	4.....k
	6.....9		7.....z	7.....z	5.....@
	7.....z				6.....9
					7.....z

Quantifiers:

We can use quantifiers to specify number of occurrences to match

- A => exactly one 'a'
- A+ => atleast one 'a'
- A* => any number of a's including zero number
- A? => atmost one a

```

import java.util.regex.*;
class RegularExpressionDemo3 {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("a");
        Matcher m = p.matcher("abaabaaabaaaab");
        while(m.find()){
            System.out.println(m.start()+"....."+m.group());
        }
    }
}

```

```
    }  
}
```

Output:

X = a	X = a+	X= a*	X= a?
0.....a	0.....a	0.....a	0.....a
2.....a	2.....aa	1.....	1.....
3.....a	5.....aaa	2.....aa	2.....a
5.....a	9.....aaaa	4.....	3.....a
6.....a		5.....aaa	4.....
7.....a		8.....	5.....a
9.....a		9.....aaaa	6.....a
10.....a		13.....	7.....a
11.....a		14.....	8.....
12.....a			9.....a
			10.....a
			11.....a
			12.....a
			13.....
			14.....

Patter class split():

We can use pattern class split method to split the target string according to a particular pattern.

```
import java.util.regex.*;  
class RegexSplitDemo  
{  
    public static void main(String[] args)  
    {  
        Pattern p = Pattern.compile("\\s");  
        String[] s = p.split("Bhupathi software solution");  
        for (String s1 :s)  
        {  
            System.out.println(s1);  
        }  
    }  
}
```

Output:

Bhupathi
software
solution

```
import java.util.regex.*;
class RegexSplitDemo
{
    public static void main(String[] args)
    {
        Pattern p = Pattern.compile("\\."); or [.]  

        String[] s = p.split("www.durgajobs.com");  

        for (String s1 :s)
        {
            System.out.println(s1);
        }
    }
}
```

Output:

www
durgajobs
com

String class split():

String class also contains split() method to split the target string according to a particular pattern

```
import java.util.regex.*;
class RegexSplitDemo1
{
    public static void main(String[] args)
    {
        String s = "bhupathi software solution";
        String[] s1 = s.split("\\s");
        for (String s2 :s1)
        {
            System.out.println(s2);
        }
    }
}
```

Output:

```
bhupathi  
software  
solution
```

note:

pattern class split method can take target string as argument where as 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

Example:

```
import java.util.regex.*;  
import java.util.StringTokenizer;  
class RegexSplitDemo2  
{  
    public static void main(String[] args)  
    {  
        StringTokenizer st = new StringTokenizer("Bhupathi software solutions");  
        while (st.hasMoreTokens())  
        {  
            System.out.println(st.nextToken());  
        }  
    }  
}
```

Output:

```
Bhupathi  
software  
solutions
```

Note:

The default regular expression for StringTokenizer is ‘space’

```

import java.util.regex.*;
import java.util.StringTokenizer;
class RegexSplitDemo3
{
    public static void main(String[] args)
    {

        StringTokenizer st = new StringTokenizer("02-06-1986","-");
        while (st.hasMoreTokens())
        {
            System.out.println(st.nextToken());
        }
    }
}

```

Write a regular expression to represent all valid 10 digit mobile numbers

Rules:

1. Every number should contain exactly 10 digits
2. The first digit should be 7, 8, 9

10 digit mobile number

[789][0-9] [0-9] [0-9] [0-9] [0-9] [0-9] [0-9]

Or

[7-9][0-9]{9}

10 – digit or 11- digit

0?[7-9][0-9]{9}

10- digit or 11-digit or 12- digit

(0|91)?[7-9][0-9]{9}

Write a regular expression to represent all valid mail id

[a-zA-Z0-9] [a-zA-Z0-9_.]* @ [a-zA-Z0-9]+ ([.][a-zA-Z]+) +

Only gmail id's

[a-zA-Z0-9] [a-zA-Z0-9_.]* @ gmail[.]com

Write a regular expression to represent all yava language identifiers

Rules:

1. Allowed characters are “ a to z” “A-Z”, “ 0-9”, “#”, “\$”
2. Length of each identifier should be at least 2 length
3. The first character should be lower case alphabet symbol for “a to k”
4. Second character should be a digit divisible by 0, 3, 6, 9

[a-k][0369][a-zA-Z0-9#\\$]*

Write a program to check whether the given number is a valid mobile number or not

```
import java.util.regex.*;
class Valid10DigitNumber
{
    public static void main(String[] args)
    {
        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");
        }
    }
}
```

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

```
import java.util.regex.*;
class ValidMailId
{
    public static void main(String[] args)
    {
        Pattern p = Pattern.compile("[a-zA-Z0-9][a-zA-Z0-9_.]*@[a-zA-Z0-9]+([.][a-zA-Z]+)+");
        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");
        }
    }
}
```

Write a program to read all mobile numbers present in the given input file where mobile numbers are mixed with normal mixed with normal text data.

```
import java.io.*;
import java.util.regex.*;
class ValidNumberMixedInput
{
    public static void main(String[] args) throws Exception
    {
        PrintWriter pw = new PrintWriter("output.txt");
        Pattern p = Pattern.compile("(0|91)?[7-9][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())
            {
                pw.println(m.group());
            }
            line = br.readLine();
        }
        pw.flush();
    }
}
```

Write a program to extract all mail id's present in the given input file where mail id's mixed with normal text data

In the above program we have replace mobile number regular expression with mail id regular expression

```
import java.io.*;
import java.util.regex.*;
class ExtractMailIds {
    public static void main(String[] args) throws Exception {
        PrintWriter pw = new PrintWriter("output.txt");
        Pattern p = Pattern.compile("[a-zA-Z0-9][a-zA-Z0-9_.]*@[a-zA-Z0-9]+([.][a-zA-Z]+)+");
        BufferedReader br = new BufferedReader(new FileReader("input.txt"));
        String line = br.readLine();
        while (line != null){
            Matcher m = p.matcher(line);
            while (m.find()){
                pw.println(m.group());
            }
            line = br.readLine();
        }
        pw.flush();
    }
}
```

Write a program to display all txt filenames present in d:\java programmes

```
import java.io.*;
import java.util.regex.*;
class DisplayAllTxtFiles
{
    public static void main(String[] args) throws Exception
    {
        int count = 0;
        Pattern p = Pattern.compile("[a-zA-Z0-9_.]+[.]txt");
        File f = new File("d:\\java programmes");
        String[] s = f.list();
        for (String s1 : s)
        {
            Matcher m = p.matcher(s1);
            if(m.find() && m.group().equals(s1))
            {
                count++;
                System.out.println(s1);
            }
        }
        System.out.println("the total number: " + count);
    }
}
```

Output:
abc.txt
abc123.txt
delete.txt
file1.txt
file2.txt
file3.txt
input.txt
output.txt
output1.txt
the total number: 9

Collections

An array is an indexed collection of fixed number of homogeneous elements. The main advantage arrays is we can represent multiple values by using single variables so that readability of the code will be improved.

Limitations of arrays:

1. Arrays are fixed in size that is once we creates an array there is no chance of increasing or decreasing the size base on over requirement. Due to this, to use arrays concept compulsory we should know the size in advance which may not possible always.
2. Array can hold only homogeneous data type elements

Example:

```
Student[] s = new Student[10000];
S[0] = new Student(); valid
S[1] = new Customer(); CE: incompatible types found: customer. Required:
student
```

3. We can solve this problem by using object type arrays

Example:

```
Object[] a = new Object[10000];
A[0] = new Student(); valid
A[1] = new Customer(); valid
```

4. Arrays concept is not implemented based on some standard data structure and hence readymade method support is not available for every requirement we have to write the code explicitly which increases complexity of programming.

To overcome above problems of arrays we should go for collection concept.

Collections are growable in nature that is based on our requirement we can increase or decrease the size

Collections can hold both homogenous and heterogeneous objects

Every collection class is implemented based on some standard data structure hence for every requirement readymade support is available. Being a programmer, we are responsible to use those methods and we are not responsible to implement those methods

Difference between arrays and collections

Arrays	Collections
1. Arrays are fixed in size that is once we create an array we can't increase or decrease the size based on our requirement.	1. Collections are growable in nature that is based on our requirement we can increase or decrease the size.
2. With respect to memory arrays are not recommended to use.	2. With respect to memory collections are recommended to use.
3. With respect to performance arrays are recommended to use.	3. With respect to performance collections are not recommended to use.
4. Arrays can hold only homogenous data type elements.	4. Collections can hold both homogenous and heterogeneous elements.
5. There is no underlying data structures for arrays and hence ready-made method support is not available. For every requirement we have to write the code explicitly which increases complexity of programming.	5. Every collection class is implemented based on some standard data structure and hence for every requirement ready-made method support is available being a programmer we can use these methods directly and we are not responsible to implement those methods.
6. Arrays can hold both primitives and objects.	6. Collections can hold only object types but not primitives.

Collection:

If we want to represent a group of individual objects as a single entity then we should go for collections.

Collection framework:

It contains several classes and interfaces which can be used to represent a group of individual objects as a single entity.

Java	C++
Collection	Container
Collection framework	STL (standard template library)

9 key interfaces of collection Framework

1. Collections
2. List
3. Set
4. SortedSet
5. NavigableSet
6. Queue
7. Map
8. SortedMap
9. NavigableMap

1. Collection(I)

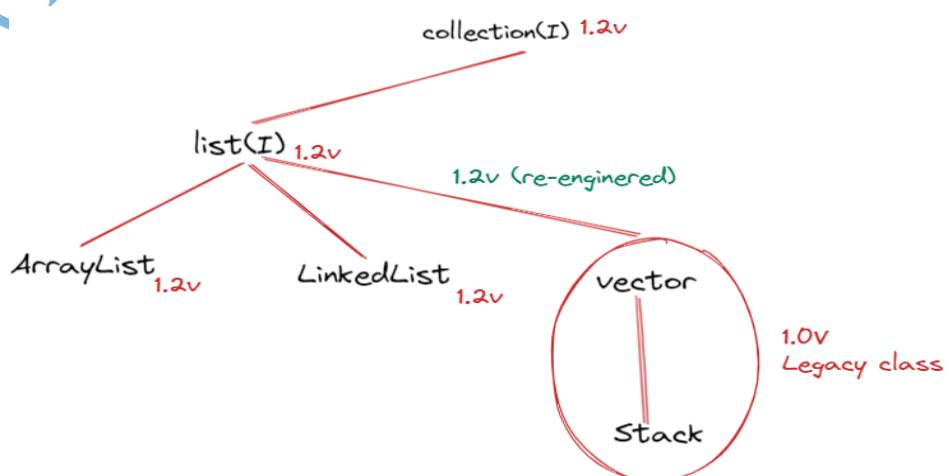
- If we want to represent a group of individual objects as a single entity then we should go for collection
- Collection interface defines the most common methods which are applicable for any collection object.
- In general collection interface is considered as root interface of collection framework.
- There is no concrete class which implements collection interface directly.

****Difference between Collection and Collections

Collection(I)	Collections(C)
Collection is an interface. If we want to represent a group of individual object as a single entity then we should go for collections	Collections is an utility class present in java.util package to define several utility methods for collection object(like sorting, searching etc...) ,

2. List(I):

- It is the child interface of collection.
- If we want to represent a group of individual objects has a single entity where duplicates are allowed and insertion order must be preserved then we should go for list

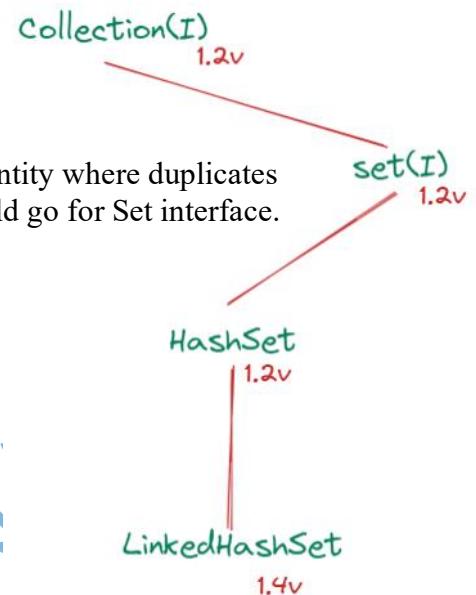


Note:

In 1.2 version vector and stack class are re- engineered to implement List interface

3. Set(I):

- It is the child interface of collection.
- If we want represent a group of individual object as a single entity where duplicates are not allowed and insertion order not required then we should go for Set interface.

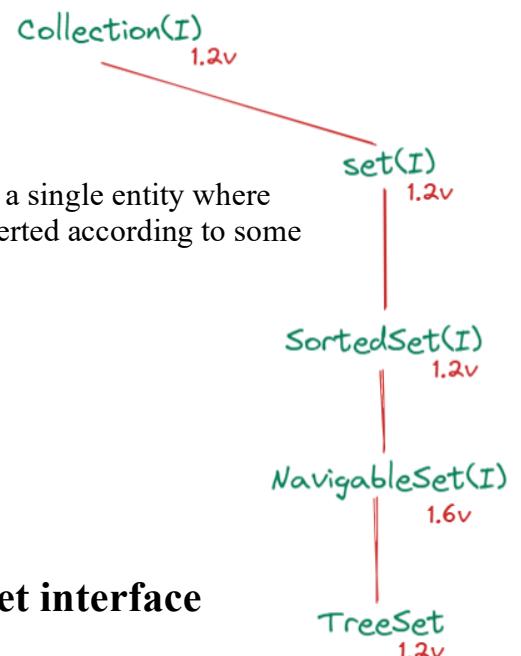


4. SortedSet(I)

- It is the child interface of set.
- If we want to represent a group of individual objects as a single entity where duplicates are not allowed and all objects should be inserted according to some sorting order then we should go for sorted set.

5. NavigableSet(I)

- It is the child interface of sorted set
- It contains several methods for navigation purposes.



***what is the difference between List and set interface

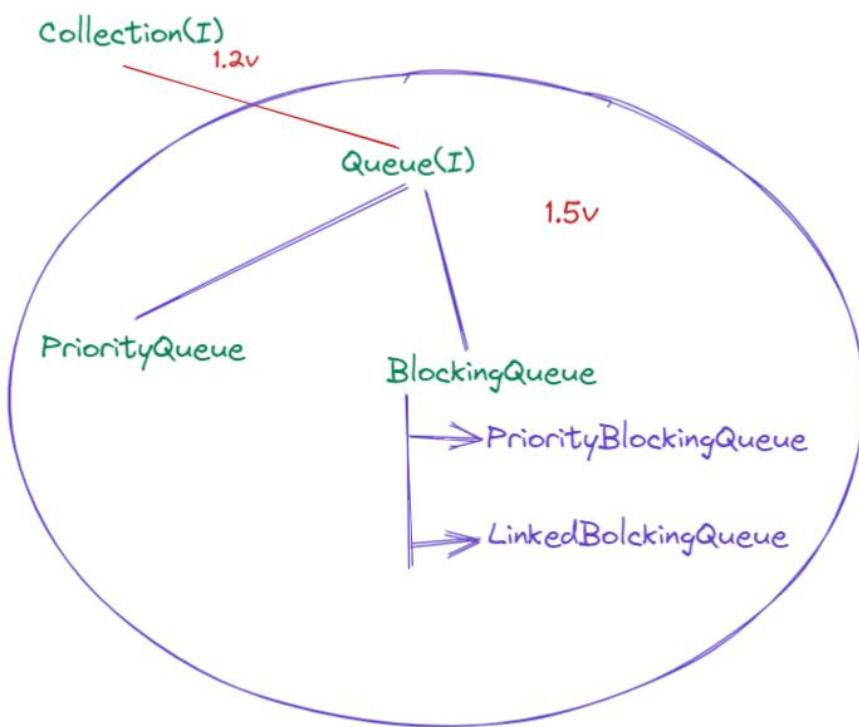
List	Set
Duplicates are allowed	Duplicates are not allowed
Insertion order preserved	Insertion order not preserved

6. Queue(I):

- It is the child interface of collection.
- If we want to represent a group of individual objects prior to processing then we should go for queue.
- Usually queue follows FIRST IN FIRST OUT order but base on our requirement we can implement our own priority order also.

Example:

Before sending a mail all mail id's we have to store in some data structure in which order we added mail ids in the same order only mail should be delivered for this requirement queue is best choice.



Note:

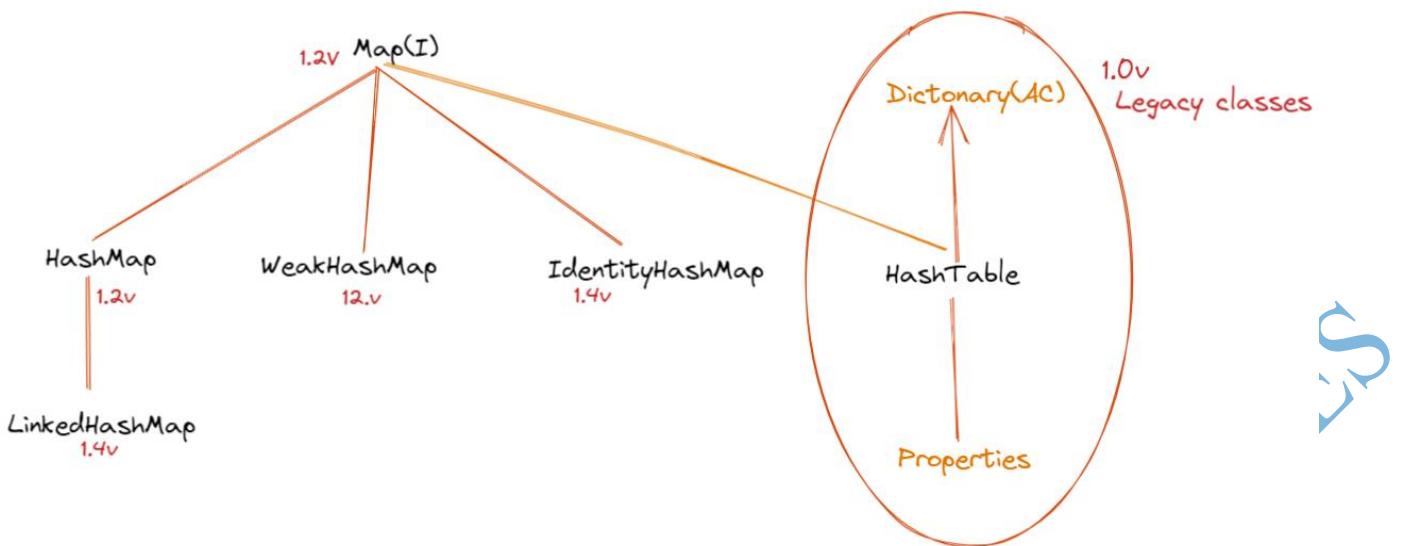
- All the above interfaces (collections, list, set, SortedSet, navigable set and queue) meant for representing a group of individual objects
- If we want to represent a group of objects as key value pair then we should go for Map.

7. Map(I)

- Map is not child interface of collection.
- If we want represent a group of objects as key value pairs then we should go for map.

Sno(key)	Name(value)
101	Durga
102	Ravi
103	shiva

- Both key values are objects only
- Duplicate key is not allowed but values can be duplicated

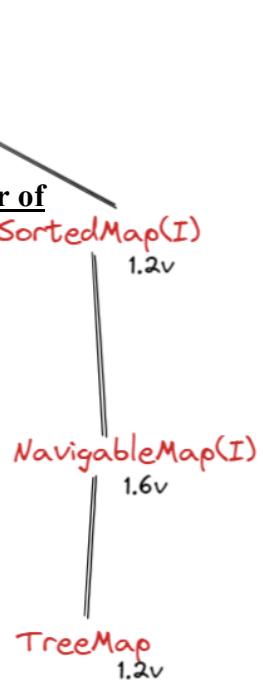


8. SortedMap(I):

- It is the child interface of map.
- If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap.
- In sorted map the sorting should be based on key but not based on value.

9. NavigableMap(I)

- It is the child interface of SortedMap
- It defines several methods for navigation purposes.

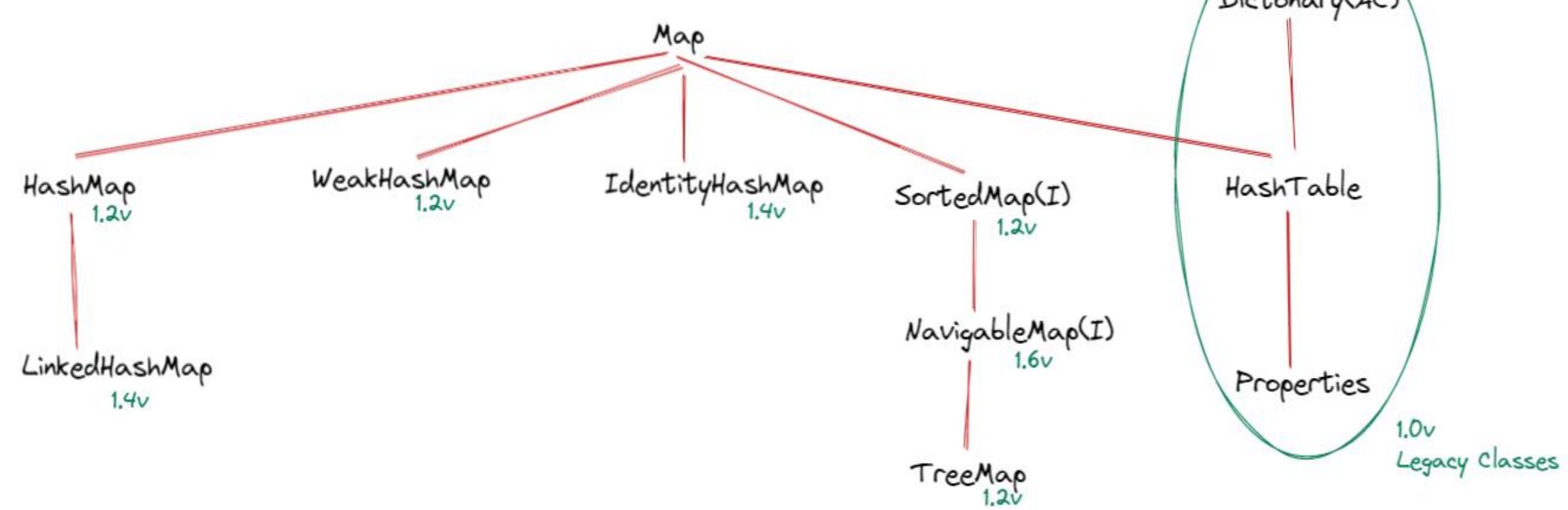


The following are legacy character present in collection framework

1. Enumeration(I)
2. Dictionary (AC)
3. Vector (C)
4. Stack (C)
5. Hashtable (C)
6. Properties (C)



VOTES



BHUPAI

Sorting

1. Comparable(I)
2. Comparator(I)

Cursors

1. Enumeration(I) 1.0v Legacy classes

2. Iterator(I)

3. ListIterator(I)

Utility classes

1. Collections

2. Arrays

Collection(I):

- If we want to represent a group of individual objects as a single entity then we should go for collections.
- Collection interface defines the most common methods which are applicable for any collection objects

Boolean add (Object o)

Boolean addAll (Collection c)

Boolean remove (Object o)

Boolean removeAll (Collection c)

To remove all objects except those present in c

Void clear ()

Boolean contains (Object o)

Boolean containsAll (Collection c)

Boolean isEmpty ()

Int size ()

Object [] toArray ();

Iterator iterator ();

Note:

There is no concrete class which implements collection interface directly

List(I):

List is child interface of collection if we want to represent a group of individual object as a single entity where duplicates are allowed and insertion order must be preserved the we should go for list

We can preserve insertion order with index and we can differentiate duplicate objects by using index. Hence index will play every important role in list.

List interface define the following specific 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 new)

```

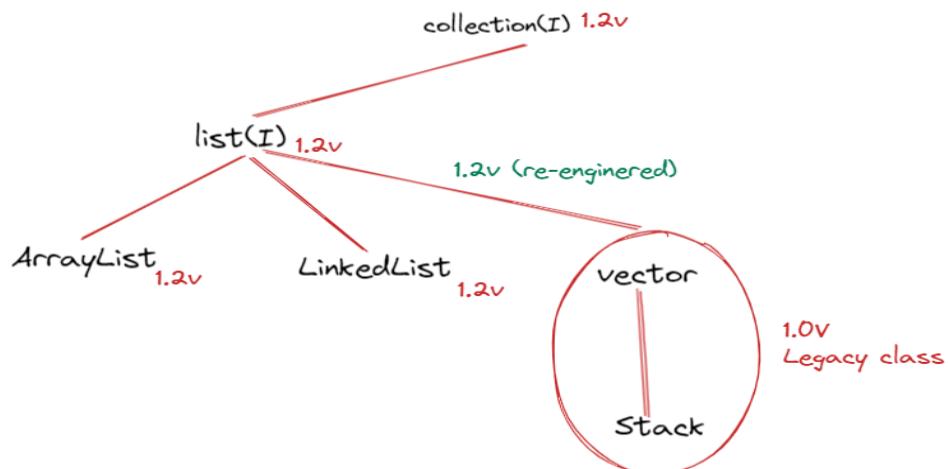
To replace the element, present at specified index with provided object and returns old object

Int indexOf(Object o)

Returns index of first occurrence of 'O'

Int lastIndexOf(Object o)

ListIterator listIterator ()



ArrayList:

- The underlying data structure is resizable array or growable array.
- Duplicates objects are allowed
- Insertion order is preserved.
- Heterogeneous objects are allowed(except treeSet and TreeMap every very heterogeneous objects are allowed)
- Null insertion is possible.

Constructors:

- ❖ `ArrayList l = new ArrayList()`
Creates an empty array list object with default initial capacity '10'.
Once array list reaches its max capacity then a new array list object will be created with

New capacity = (current capacity * 3/2)+1
- ❖ `ArrayList l = new ArrayList (int IntialCapacity)`
Creates an empty array list object with specified initial capacity.
- ❖ `ArrayList l = new ArrayList (Collection c)`
Creates an equivalent array list object for the given collection

Example:

```
import java.util.*;
class ArrayListDemp
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add("A");
        l.add(10);
        l.add("A");
        l.add(null);
        System.out.println(l);
        l.remove(2);
        System.out.println(l);
        l.add(2,"m");
        l.add("n");
        System.out.println(l);
    }
}
```

- ❖ Usually, we can use collections to hold under transfer object form one location to another location(container) to provide support for these requirement every collection class by default implements serializable and cloneable interfaces.
- ❖ ArrayList, vector classes implements random access interface so that any random element we can access with same speed.

Random Access:

Random access interface present in java.util package and it doesn't contain methods it is a market interface, where required ability will be provided automatically by the JVM.

Example:

```
import java.util.*;
import java.io.Serializable;
class RandomAccessDemo
{
    public static void main(String[] args)
    {
        ArrayList l1 = new ArrayList();
        LinkedList l2 = new LinkedList();
        System.out.println(l1 instanceof Serializable);//true
        System.out.println(l2 instanceof Cloneable);//true
        System.out.println(l1 instanceof RandomAccess);//true
        System.out.println(l2 instanceof RandomAccess);//false
    }
}
```

- ❖ ArrayList is the best choice, if our frequent operation is retrieval operation (because array list implements random access interface).
- ❖ ArrayList is the worst choice is our frequent operation is insertion, deletion for the middle.

Difference between ArrayList and vector:

ArrayList	Vector
1. Every method present in the ArrayList is non-Synchronized	1. Every method present in the vector is Synchronized.
2. At a time multiple threads are allowed to operate in array list object under hence it is not thread safe	2. At time only one thread is allowed to operate an vector object and hence it is thread safe
3. Relatively performance is high because thread are required to wait to operate on array list object.	3. Relatively performance is low because threads are required to wait to operate on vector object.
4. Introduced in 1.2 version and it is non-legacy	4. Introduced in 1.0 version and it is legacy

How to get synchronized version of ArrayList object

- ❖ By default ArrayList is non-synchronized but we can get synchronized version of ArrayList object by using “**synchronizedList()**” method of Collections class.

public static List synchronizedList (List l)

example:

```
ArrayList l = new ArrayList();
List l1 = Collections.synchronizedList(l);
```

- ❖ Similarly we can get synchronized version of set and map objects by using the following methods of Collections class.

public static Set synchronizedSet(Set s)
public static Map synchronizedMap(Map m)

LinkedList:

- The under-laying data structure is double LinkedList insertion order presented. duplicate objects are allowed, heterogeneous object are allowed, null insertion is possible.
- LinkedList implements Serializable and Cloneable interface but not random access.
- LinkedList is the best choice if our frequent operation in insertion or deletion in the middle.
- LinkedList is the worst choice if our frequent operation is retrieval operation.

Constructor:

1. `LinkedList l = new LinkedList();`
Creates an empty Linked List object
2. `LinkedList l = new LinkedList(Collection c)`
Creates an equivalent LinkedList object for the given collection.

LinkedList class specific methods:

- Usually we can use LinkedList to develop stacks and queue to provide support for these requirement LinkedList class defines the following specific methods

```
Void addFirst(Object o);
Void addLast (Object o);
Object getFirst();
Object getLast ();
Object removeFirst();
Object removeLast();
```

Example:

```
import java.util.*;
class LinkedListDemo
{
    public static void main(String[] args)
    {
        LinkedList l = new LinkedList();
        l.add("malachi");
        l.add(30);
        l.add(null);
        l.add("bhupathi");//[malachi, 30, null, bhupathi]
        l.set(0,"software");//[software, 30, null, bhupathi]
        l.add(0,"solutions");//[solutions, software, 30, null, bhupathi]
        l.removeLast();//[solutions, software, 30, null]
        l.addFirst("renuka");//[renuka, solutions, software, 30, null]
        System.out.println(l);
    }
}
```

Difference between ArrayList and LinkedList

ArrayList	LinkedList
1. ArrayList is the best choice if our frequent operation is retrieval operation	1. LinkedList is the best choice if our frequent operation is insertion or deletion in the middle
2. ArrayList is the worst choice if our frequent operation is insertion and deletion in the middle because internally several shift operations are performed.	2. LinkedList is the worst choice if our frequent operation is retrieval operation.
3. In ArrayList the elements will be stored in consecutive memory	3. In LinkedList the elements won't be stored in consecutive memory location and hence retrieval operation will become complex.

Vector:

- The underlying data structure resizable array or growable array.
- Insertion order is preserved
- Duplicates are allowed.
- Heterogeneous objects are allowed.
- Null insertion is possible.
- It implements serializable, cloneable and Random Access interface.
- Every method present in the vector is synchronized and hence vector object is thread safe.

Constructor:

1. `Vector v = new Vector();`
Creates an empty vector object with default initial capacity is 10. Once vector reaches its maximum capacity then a new vector object will be created with

$$\text{New capacity} = \text{current capacity} * 2$$

2. `Vector v = new Vector(int InitialCapacity);`
Creates an empty vector object with specified initial capacity.

3. `vector v = new Vector(int InitialCapacity, int IncrementalCapacity)`
`V = new v(1000,5);`

4. `vector v = new Vector(Collection c);`
Creates an equivalent vector object for the given collection. This constructor is meant for interconversion between Collection Objects

Vector specific methods:

To add objects

Add (Object o) → Collections
Add (int index, Object o) → List
AddElement (Object o) → Vector

To remove objects

Remove (Object o) → C
RemoveElement (Object o) → V
Remove(int index) → L
removeElementAt(int index) → V
clear() → C
removeAllElements() → V

to get Objects

Object get(int index) → L
Object elementAt(int index) → V
Object firstElement() → V
Object lastElement() → V

Other methods

Int size();
Int capacity();
Enumeration elements()

Example:

```
import java.util.*;
class VectorDemo
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        System.out.println(v.capacity());//10

        for (int i= 1;i<=10 ;i++ )
        {
            v.addElement(i);
        }
        System.out.println(v.capacity());//10
        v.addElement("A");
        System.out.println(v.capacity());//20
        System.out.println(v);//[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A]
    }
}
```

Stack:

- It is a child class of vector
- It is a specially designed class for **Last in First Out Order (LIFO)**

constructor

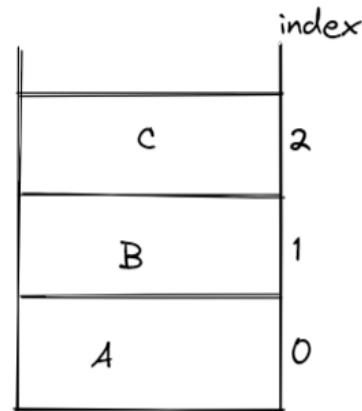
```
Stack s = new Stack();
```

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
Boolean empty()
    Returns true if the stack is empty
Int search(Object o)
    Returns offset if the element is available otherwise return -1
```

Example:

```
import java.util.*;
class StackDemo
{
    public static void main(String[] args)
    {
        Stack s = new Stack();
        s.push("A");
        s.push("B");
        s.push("C");
        System.out.println(s); // [A, B, C]
        System.out.println(s.search("A")); // 3
        System.out.println(s.search("z")); // -1
    }
}
```



The 3 cursors of java

If we want get objects one by one from the collection then we should go for cursor.
There are three types of cursor available in java

1. Enumeration
2. Iterator
3. ListIterator

Enumeration

- We can use enumeration to get objects one by one form legacy collection object.
- We can create enumeration object by using elements method of vector class.

```
Public Enumeration elements();
```

Example:

Enumeration e = v. elements ();
V → vector

methods

public boolean hasMoreElements();
public Object nextElement();

```
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, 10]  
        Enumeration e = v.elements();  
        while(e.hasMoreElements())  
        {  
            Integer i = (Integer)e.nextElement();  
            if(i%2==0)  
            {  
                System.out.println(i);  
            }  
            else  
            {  
                System.out.println(i+"will be removed");  
                v.remove(i);  
                System.out.println(v);  
            }  
        }  
        System.out.println(v);  
    }  
}
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
0  
1will be removed  
[0, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
3will be removed  
[0, 2, 4, 5, 6, 7, 8, 9, 10]  
5will be removed  
[0, 2, 4, 6, 7, 8, 9, 10]
```

```
7will be removed  
[0, 2, 4, 6, 8, 9, 10]  
9will be removed  
[0, 2, 4, 6, 8, 10]  
[0, 2, 4, 6, 8, 10]  
*/
```

Limitation of enumeration:

- We can apply enumeration concept only for legacy class and it is not a universal cursor.
- By using enumeration, we can get only read access and we can't perform remove operation.
- To overcome above limitations, we should go for iterator.

Iterator(I)

- We can apply iterator concept for any collection object and hence it is universal cursor.
- By using iterator we can perform both read and remove operations.
- We can create iterator object by using iterator method of collection interface

Public Iterator iterator()

Example:

```
Iterator itr = c.iterator();  
C - any collection object
```

Methods:

```
Public boolean hasNext()  
Public Object next()  
Public void remove()
```

Example:

```
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, 10]  
        Iterator itr = l.iterator();  
        while (itr.hasNext())  
        {  
            Integer I = (Integer)itr.next();  
            if (I%2 == 0)
```

```

        {
            System.out.println(l);
        }
    }
}
/*
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
0
2
4
6
8
10
[0, 2, 4, 6, 8, 10]
*/

```

Limitations of iterator

- By using enumeration and iterator we can always move only words forward direction and we can't move towards backward direction these are single direction cursors but not bidirectional cursor
- By using iterator we can perform only read and remove operations and we cannot perform replacement and addition of new objects
- To overcome above limitations, we should go for listIterator

ListIterator(l):

- By using listIterator we can move either to the forward direction or to the backward direction and hence it is a bidirectional cursor
- By using list iterator we can perform replacement and addition of new objects in addition to read and remove operation.
- We can create list iterator by using listIterator method of list Interface.

Public ListIterator listIterator ();

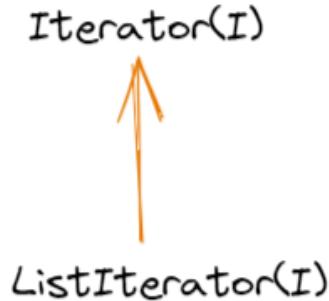
Example:

 ListIterator ltr = l.listIterator();

L - ArrayList Object

Methods:

- ListIterator is the child interface of iterator and hence all methods present in iterator by default available to the ListIterator



- List iterator is define the following 9 methods

Forward movement

```
Public boolean hasNext()  
Public Object next()  
Public int nextIndex()
```

Backward movement

```
Public boolean hasPrevious();  
Public Object previous();  
Public int previousIndex();
```

Extra operations

```
Public void remove();  
Public void add(Object o)  
Public void set(Object o);
```

Example:

```
import java.util.*;  
class ListIteratorDemo  
{  
    public static void main(String[] args)  
    {  
        LinkedList l = new LinkedList();  
        l.add("bhupathi");  
        l.add("renuka");  
        l.add("issacraj");  
        l.add("jeswica");  
        System.out.println(l); // [bhupathi, renuka, issacraj, jeswica]
```

```

ListIterator ltr = l.listIterator();
while (ltr.hasNext())
{
    String s = (String) ltr.next();
    if (s.equals("jeswica"))
    {
        ltr.remove();
    }
    else if (s.equals("issacraj"))
    {
        ltr.add("chintu");
    }
    else if (s.equals("bhupathi"))
    {
        ltr.set("malachi");
    }
}
System.out.println(l); // [malachi, renuka, issacraj, chintu]
}
}

```

- The most powerful cursor is ListIterator but its limitation is it applicable only for List Object

Comparison table of 3 cursor

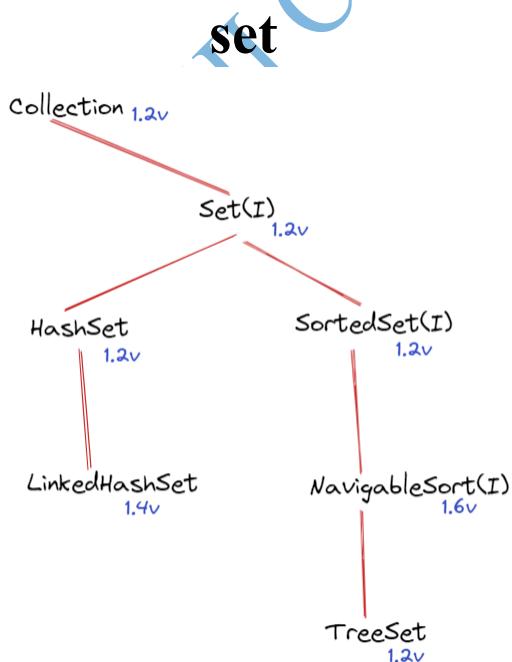
Property	Enumeration	Iterator	ListIterator
Where we can apply	Only for legacy classes	For any collection Object	Only for list Object
It is legacy?	Yes (1.0 version)	No (1.2 version)	No (1.2 version)
Movement	Single direction (only forward direction)	Singe direction (only forward direction)	Bidirectional
Allowed operational	Only read	Read or remove	Read or remove Replace or add
How we can get	By using element () of vector class	By using Iterator() of collection (I)	By using listIterator () of List(I)
Methods	2 methods hasNextElements () nextElement ()	3 methods hasNext() next() remove()	9 methods hasNext () next () nextIndex() boolean hasPrevious (); Object previous (); int previousIndex ();

Internal implementation of cursor

```
import java.util.*;
class CursorDemo
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        Enumeration e = v.elements();
        Iterator itr = v.iterator();
        ListIterator ltr = v.listIterator();
        System.out.println(e.getClass().getName());
        System.out.println(itr.getClass().getName());
        System.out.println(ltr.getClass().getName());
    }
}
```

output:

```
java.util.Vector$1
java.util.Vector$Itr
java.util.Vector$ListItr
```



- Set is child interface of collection
- If we want to represent a group of individual object as a single entity where duplicates are not allowed and insertion order not preserved.
- Set interface doesn't contain any new method and we have to use only collection interface method

HashSet:

- The underlying data structure is Hashtable.
- Duplicate objects are not allowed
- Insertion order is not preserved and it is based on hashCode of objects null insertion possible (only once)
- Heterogenous objects are allowed.
- Implements serializable and cloneable but not random-access interface
- HashSet is the best choice if our frequent operation is search operation

Note:

*** in HashSet duplicates are not allowed if we are trying to insert duplicates then we won't get any compile time or runtime errors and add methods simply returns false

```
import java.util.*;
class HashSetDemo1
{
    public static void main(String[] args)
    {
        HashSet h = new HashSet ();
        System.out.println(h.add("A")); //true
        System.out.println(h.add("A")); //false
    }
}
```

Constructor:

HashSet h = new HashSet();

Creates an empty HashSet object with default initial capacity 16 and default fill ration = 0.75

HashSet h = new HashSet(int initialcapacity);

Creates an empty HashSet object with specified intial capacity and default fill ratio 0.75

HashSet h = new HashSet(int intialcapacity, float fillRatio)

HashSet h = new hashSet(Collection c)

Creates an equivalent HashSet for the given collection.

This constructor meant for inter conversation between collection and objects

Fill ratio / loadfactor:

- After filling how much ratio a new HashSet object will be created, this ratio is called fill Ratio or load factor.

Example:

Fill jratio = 0.75 means after fill 75% ratio an new HashSet object will be created.

Example:

```
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, B, C, D, Z, 10]
    }
}
```

LinkedHashSet:

- It is the child class of HashSet
- It is exactly same as HashSet (Including Constructors and methods) except the following difference

HashSet	LinkedHashSet
The underlying data structure is Hashtable	Underlaying data structure is a combination of linked list and Hashtable
Insertion order not preserved	Insertion order preserved
Introduced in 1.2 version	Introduced in 1.4 version

Note:

in the above program if we replace HashSet with LinkedHashSet then output is [B, C, D, Z, null, 10] that is insertion order preserved

Example:

```
import java.util.*;
class LinkedHashSetDemo {
    public static void main(String[] args) {
        LinkedHashSet h = new LinkedHashSet();
        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); // [B, C, D, Z, null, 10]
    }
}
```

Note:

In general, we can use LinkedHashSet to develop cache-based application where duplicates are not allowed and insertion order preserved.

SortedSet(I)

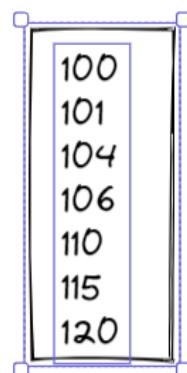
- SortedSet is the child Interface of set if we want to represent a group of individual objects according to some sorting order without duplicates than we should go for SortedSet
- SortedSet interface defines the following specific methods
 1. Object first()
Returns first element of the SortedSet
 2. Object last()
Returns last element of the SortedSet
 3. SortedSet headSet(Object obj)
Returns SortedSet whose elements are less than obj
 4. SortedSet tailSet(Object obj)
Returns SortedSet whose elements are \geq obj
 5. SortedSet(Object obj1, Object obj2)
Returns SortedSet whose elements are \geq obj1 and obj2
 6. Comparator Comparator();
Returns Comparator Object that describes underlying Sorting technique. If we are using default natural sorting order the we will get null.

Note:

The default natural sorting order for numbers is ascending order and for string objects is alphabetical order

Example:

1. First() => 100
2. Last() => 120
3. headSet(106) => [100,101,104]
4. tailSet(106) => [106,110,115,120]
5. subset(101,115) => [101,104,106,110]
6. Comparator() => null



TreeSet

- The underlaying data structure is **Balanced Tree**
- Duplicate object are not allowed
- Insertion order not preserved
- Heterogeneous objects are not allowed other wise we will get runtime exception saying ClassCastException(CCE)
- Null exception is possible(only once)
- TreeSet implements serializable and cloneable but not random access
- All object will be inserted based on some sorting order it may be default natural sorting order or customized sorting order

constructor

1. `TreeSet t = new TreeSet();`
Creates is an empty TreeSet object where the elements will be inserted according to default natural sorting ordered.
2. `TreeSet t = new TreeSet (comparator c)`
Creates an empty TreeSet object where the elements will be inserted according to customized sorting order specified by comparator object.
3. `TreeSet t = new TreeSet (Collection c)`
4. `TreeSet t = new TreeSet (SortedSet s)`

```
import java.util.*;
class TreeSetDemo
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet();
        t.add("A");
        t.add("a");
        t.add("B");
        t.add("Z");
        t.add("L");
        //t.add(new Integer(10));
                    //java.lang.ClassCastException: java.lang.String cannot
                    //be cast to java.lang.Integer
        //t.add(null); //java.lang.NullPointerException
        System.out.println(t);//[A, B, L, Z, a]
    }
}
```

Null Acceptance:

- For non-empty TreeSet if we are trying to insert “null” then we will get “NullPointerException”.
- For empty TreeSet as first element null is allowed but after inserting that null, if we are trying to insert to any other then we will get Runtime Exception saying “NullPointerException”.

Note:

*** until 1.6 version null is allowed as a first element to the empty TreeSet but from 1.7 version onwards null is not allowed even as the first Element “null” such type of stories not applicable for TreeSet from 1.7 version on wards.

```
import java.util.*;
class TreeSetDemo1
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet();
        t.add(new StringBuffer("A"));
        t.add(new StringBuffer("a"));
        t.add(new StringBuffer("B"));
        t.add(new StringBuffer("Z"));
        t.add(new StringBuffer("L"));
        System.out.println(t);
        //java.lang.ClassCastException: java.lang.StringBuffer cannot
        be cast to java.lang.Comparable
    }
}
```

- If we are depending on default natural sorting order compulsory the object should be homogeneous and comparable otherwise, we will get runtime exception saying “ClassCastException”.
- An object is set to be comparable if and only if corresponding class implements comparable interface.
- String class and all wrapper class already implement comparable interface but StringBuffer class doesn't implement comparable interface hence we got “ClassCastException” in the above example

Comparable(I) interface:

- It is present in `java.lang` package and it contains only one method “`compareTo()`”

Public int CompareTo(Object obj);

`obj1.compareTo(obj2)`

returns -Ve

iff `obj1` has to come before `obj2`

returns +Ve

iff `obj1` has to come after `obj2`

return 0

iff `obj1` and `obj2` are equal

Example:

```
import java.util.*;
class CompareToSDemo
{
    public static void main(String[] args)
    {
        System.out.println("A".compareTo("Z"));//-ve
        System.out.println("z".compareTo("A"));//+ve
        System.out.println("A".compareTo("A"));//0
        System.out.println("A".compareTo(null));//java.lang.NullPointerException
    }
}
```

- If we are depending on default natural sorting order the while adding object into the TreeSet JVM will call compareTo() method.

Example:

```
import java.util.*;
class CompareToDemo1
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet();
        t.add("k");
        t.add("Z");
        t.add("A");
        t.add("A");
        System.out.println(t);//[A, Z, k]
    }
}
```

$\boxed{\text{obj1}}.\text{compareTo}(\boxed{\text{obj2}})$

the Object,
which is to be inserted

the Object,
which is already inserted

Note:

If default natural sorting order not available or if we are not satisfied with default natural sorting order the we can go for customization sorting by using comparator.

Comparable meant for **default natural sorting order** where as
Comparator meant for **customized sorting order**

Comparator:

- Comparator present in `java.util` package and it defines two methods “`compare()` and `equals()`”
 - ✓ Public in `compare(Object obj1, Object obj2)`
 - ⇒ Returns -ve if obj1 has to come before obj2
 - ⇒ Returns +ve if obj1 has to come after obj2
 - ⇒ Returns 0 if obj1 and obj2 are equal
 - ✓ Public boolean `equal(Object obj)`
- When ever we are implementing comparator interface compulsory we should provide implementation only for `compare()` method and we are not require to provide implementation for `equals` method because it is all ready available to our class from object class through inheritance.

Write a program to insert integer objects into the 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()); → line 1
        t.add(10);
        t.add(0);      => +ve compare(0,10)

        t.add(15);     => -ve compare(15,10)

        t.add(5);      => +ve compare(5,10)
                        => -ve compare(5,0)

        t.add(20);     => -ve compare(20,10)
                        => +ve compare(20,15)

        t.add(20);     => -ve compare(20,10)
                        => -ve compare(20,15)
                        => 0 compare(20,20)
```

```

        System.out.println(t);/[20, 15, 10, 5, 0]
    }
}
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        Integer I1 = (Integer)obj1;
        Integer I2 = (Integer)obj2;
        if(I1<I2)
        {
            return +1;
        }
        else if (I1>I2)
        {
            return -1;
        }
        else
            return 0;
    }
}

```

- Line 1 if we are not passing comparator object then internally JVM will call “compareTo ()” method which is meant for “default natural sorting order” in this case the output is [0,5,10,15,20]
- At line 1 if we are passing comparator object the JVM will call “compare()” method which is meant of customized sorting in this case output is [20,15,10,5,0].

Various possible implement of compare() method:

```

Public int compare(Object obj1, Object obj2){
{
    Integer I1 = (Integer)obj1;
    Integer I2 = (Integer)obj2;
    1. Return I1.compareTo(I2); default natural sorting order [ascending
       order][0,5,10,15,20]
    2. Return -I1.compareTo(I2); [descending order][20,15,10,5,0]
    3. Return I2.compareTo(I1); [descending order][20,15,10,5,0]
    4. Return -I2.compareTo(I1); [ascending order][0,5,10,15,20]
    5. Return +1; [Insertion order][10,0,15,5,20,20]
    6. Return -1; [reverse of insertion order][20,20,5,15,0,10]
    7. Return 0;[only first element will be inserted and all remaining are consider as
       duplicate][10]
}

```

Write a program to insert String object into the treeSet where all elements should be inserted according to reverse of alphabetical order

```
import java.util.*;
class TreeSetDemo2
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet(new MyComparator());
        t.add("bhupathi");
        t.add("malachi");
        t.add("renuka");
        t.add("issacraj");
        t.add("jeswica");
        t.add("akshitha");
        System.out.println(t);
    }
}
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = (String)obj1;
        String s2 = (String)obj2;
        return s2.compareTo(s1);
        // [renuka, malachi, jeswica, issacraj, bhupathi, akshitha]
        return -s1.compareTo(s2);
        // [renuka, malachi, jeswica, issacraj, bhupathi, akshitha]
        return s1.compareTo(s2);
        // [akshitha, bhupathi, issacraj, jeswica, malachi, renuka]
    }
}
```

Write a program to insert StringBuffer object into TreeSet where sorting order is alphabetical order

```
import java.util.*;
import java.lang.*;
class TreeSetDemo4 {
    public static void main(String[] args) {
        TreeSet t = new TreeSet(new MyComparator());
        t.add(new StringBuffer("b"));
        t.add(new StringBuffer("m"));
        t.add(new StringBuffer("r"));
        t.add(new StringBuffer("i"));
        t.add(new StringBuffer("j"));
        t.add(new StringBuffer("a"));
        System.out.println(t);
    }
}
```

```

class MyComparator implements Comparator
{
    public int compare (Object obj1, Object obj2)
    {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s1.compareTo(s2); // [a,b,i,j,m,r]
    }
}

```

Note:

- If we are depending an default natural sorting order compulsory object should be homogeneous and comparable otherwise we will get runtime exception saying “`ClassCastException`”.
- If we are defining our own sorting by comparator then objects need not be comparable and homogeneous that is we can add heterogeneous non comparable objects also.

Write a program to insert sorting and StringBuffer objects into TreeSet where sorting order is increasing length order. If two objects having same length then consider their alphabetical order.

```

import java.util.*;
class TreeSetDemo5
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet(new MyComparator());
        t.add("A");
        t.add(new StringBuffer("ABCD"));
        t.add(new StringBuffer("AA"));
        t.add("XXX");
        t.add("ABCDE");
        t.add("A");
        System.out.println(t); // [A, AA, XXX, ABCD, ABCDE]
    }
}
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
    }
}

```

```

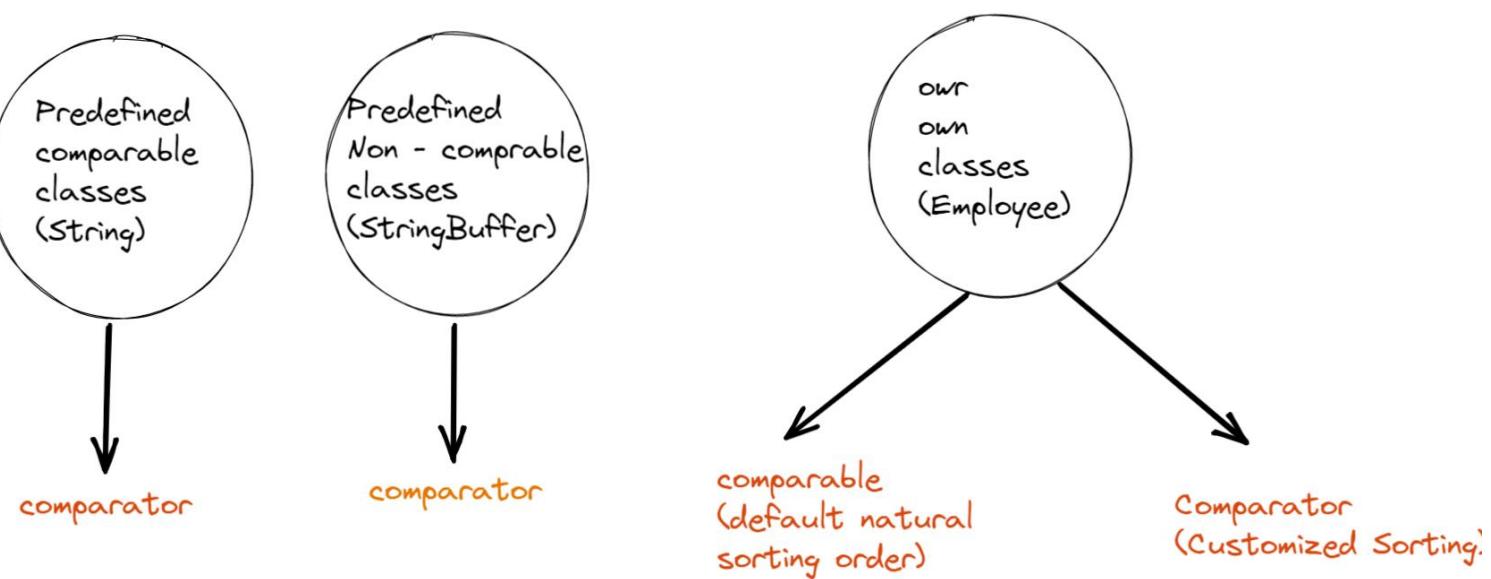
        return s1.compareTo(s2);
    }
}

```

Comparable V/S comparator:

- For predefined comparable classes default natural sorting order already available if we are not satisfied with that default natural sorting order then we can define our own sorting using comparator.
- For predefined non comparable classes (like StringBuffer) default natural sorting order not already available we can define our own sorting by using comparator.
- For our own classes like employee, then person who is writing the class is responsible to define default natural sorting order by implementing comparable interface

The person who is using our class, if he is not satisfied with default natural sorting order then he can define his own sorting comparator



```

import java.util.*;
class Employee implements Comparable
{
    String name;
    int eid;
    Employee(String name, int eid)
    {
        this.name = name;
        this.eid = eid;
    }
    public String toString()
    {
        return name+"----"+eid;
    }
    public int compareTo(Object obj)
    {

```

```

        int eid1 = this.eid;
        Employee e =(Employee)obj;
        int eid2 = e.eid;
        if(eid1<eid2)
            return -1;
        else if (eid1>eid2)
            return +1;
        else
            return 0;
    }
}

class ComparatorVSComparable
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee("malachi",100);
        Employee e2 = new Employee("bhupathi",200);
        Employee e3 = new Employee("renuka",50);
        Employee e4 = new Employee("issacraj",150);
        Employee e5 = new Employee("jeswica",100);

        TreeSet t = new TreeSet();
        t.add(e1);
        t.add(e2);
        t.add(e3);
        t.add(e4);
        t.add(e5);

        System.out.println(t); // [renuka----50, malachi----100, issacraj----150,
        bhupathi----200]

        TreeSet t1 = new TreeSet(new MyComparator());
        t1.add(e1);
        t1.add(e2);
        t1.add(e3);
        t1.add(e4);
        t1.add(e5);

        System.out.println(t1); // [bhupathi----200, issacraj----150, jeswica----100,
        malachi----100, renuka----50]
    }
}

class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        Employee e1 = (Employee)obj1;
        Employee e2 = (Employee)obj2;
    }
}

```

```

        String s1 = e1.name;
        String s2 = e2.name;
        return s1.compareTo(s2);
    }

}

```

Comparison of Comparable and Comparator:

Comparable	Comparator
• It is meant for default natural sorting order	• It is meant for customized sorting order
• Present java.lang package	• Present in java.util package
• It defines only one method “compareTo()”	• It defines two methods “compare()”, and “equals()”.
• Sorting and wrapper classes implements comparable interface	• The only implemented classes of comparator Collator and RuleBased Collator

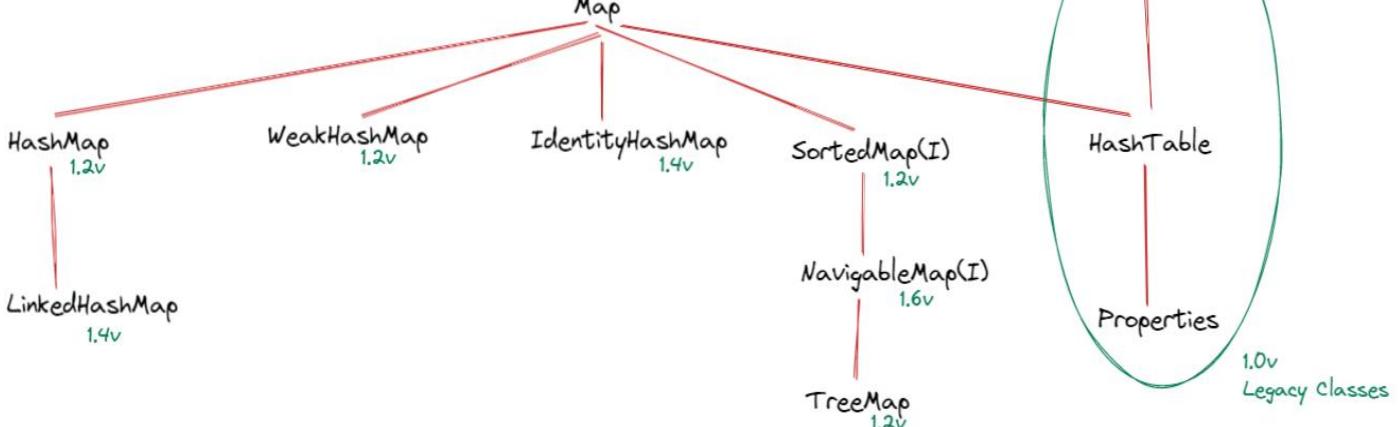
Comparison table of set implemented class

Property	HashSet	LinkedHashSet	TreeSet
1. Underlying data structure	Hashtable	Combination of LinkedList + Hashtable	Balance Tree
2. Duplicate objects	Not allowed	Not allowed	Not allowed
3. Insertion order	Not preserved	Preserved	Not preserved
4. Sorting order	Not applicable	Not applicable	applicable
5. Heterogeneous	Allowed	Allowed	Not allowed
6. Null acceptance	Allowed	Allowed	For empty TreeSet as first element null is allowed

Note:

For empty TreeSet as the first element Null is allowed but this rule is applicable until 1.6 version only for 1.7 version onwards null is not allowed even as the first element.

Map(I)



- Map (I) is not child interface of collection.
- If we want to represent a group objects has a key value pairs then we should go for map.

Key	value
101	Malachi
102	Renuka
103	Bhupathi
104	Issacraj
105	Jeswica

- Both the key and values are objects only
- Duplicate keys are not allowed but values can be duplicated.
- Each key value pair is called “entity” hence map is consider as a collection of entry objects.

Map(I) methods

1. Object put(Object key, Object value)

To add one key value pair to the map, if the key already present then old value will be replaced with new value and returns old values.

example:

```

null   m.put(101, "malachi")
      m.put(102, " bhupathi")
      malachi m.put(101, "renuka")
  
```

```

      {   renuka
          101 - malachi
          102 shiva
      }
  
```

2. Void putAll (Map m);

3. Object get (Object key)

Returns the value associated with specified key

4. Object remove (Object key)

Removes the entry associated with specified key

- 5. Boolean containsKey (Object key);
 - 6. Boolean containsValue (Object value);
 - 7. Boolean isEmpty ()
 - 8. Int size ();
 - 9. Void clear ()

 - 10. Set keyset()
 - 11. Collection values()
 - 12. Set entrySet()
- }
- Collection views of map

Entry(I)

A map is a group of key value pairs and each key value pair is called an entry. Hence map is considered as a collection of entry objects, without existing map object there is no chance of existing entry object hence, Entry(I) is define inside Map interface.

Interface Map

```
{
    Interface Entry
    {
        Object getKey();
        Object getValue();
        Object setValue(Object new);
    }
}
```

}

Entry specific method and we can apply only on entry object

HashMap:

- The under laying data structure is HashTable
- Insertion order is not preserved and it is based on hashCode of key's
- Duplicate key's are not allowed but values cab be duplicated.
- Heterogeneous objects are allowed for both key and value
- Null is allowed for key(only once)
- Null is all for values(any number of times)
- HashMap implements serializable and cloneable interfaces but not Rando Access.
- HashMap is the best choice if our frequent operation is search operation.

Constructor:

1. `HashMap m = new HashMap ()`
Creates an empty HashMap Object with default intialCapacity is 16 and default fillRatio is 0.75
2. `HashMap m= new HashMap (int intialCapacity);`
Creates an empty HashMap object with specified intialCapacity and default fillRatio 0.75.
3. `HashMap m = new HashMap (int InitialCapacity, float fillratio)`
4. `HashMap m = new HashMap (Map m)`

```

import java.util.*;
class HashMapDemo
{
    public static void main(String[] args)
    {
        HashMap m = new HashMap();
        m.put("malachi",700);
        m.put("Renuka",800);
        m.put("issacraj",200);
        m.put("jeswica",500);

        System.out.println(m);
            //{{Renuka=800, jeswica=500, malachi=700, issacraj=200}
        System.out.println(m.put("malachi",1000));//700

        Set s = m.keySet();
        System.out.println(s);//[Renuka, jeswica, malachi, issacraj]

        Collection c = m.values();
        System.out.println(c);//[800, 500, 1000, 200]

        Set s1 = m.entrySet();
        System.out.println(s1);
            //[[Renuka=800, jeswica=500, malachi=1000, issacraj=200]

        Iterator itr = s1.iterator();
        while (itr.hasNext())
        {
            Map.Entry m1 = (Map.Entry)itr.next();
            System.out.println(m1.getKey()+"----"+m1.getValue());
                /*Renuka----800
                jeswica----500
                malachi----1000
                issacraj----200*/
            if (m1.getKey().equals("malachi"))
            {
                m1.setValue(10000);
            }
        }
        System.out.println(m);
            {Renuka=800, jeswica=500, malachi=10000, issacraj=200}
    }
}

```

Difference between HashMap and Hashtable

HashMap	Hashtable
• Every method present in HashMap is Not synchronized	• Every method present in Hashtable is synchronized.
• At a time multiple threads allow to operator on HashMap object and hence it is not thread safe	• At a time only one thread is allowed to operator on Hashtable and hence it is thread safe
• Relatively performance is high because thread are not required to wait to operate on HashMap object	• Relatively performance is low because threads are required to wait to operation on Hashtable object
• Null is allowed for both the key and value	• Null is not allowed for keys and values otherwise we will get null pointer exception.
• Introduced in 1.2 version and it is not legacy	• Introduced in 1.0 version and it is legacy

How to get synchronized version of HashMap object

By default HashMap is non synchronized but we can get synchronized version of HashMap by using synchronizedMap() of Collection class

```
HashMap m = new HashMap();
Map m1 = Collections.synchronizedMap(m);
```

LinkedHashMap:

- It is the child class of HashMap
- It is exactly same as HashMap (Including methods and constructors) except the following difference.

HashMap	LinkedHashMap
• The underlying data structure is HashTable	• Underlying data structure is a combination of LinkedList and Hashtable (Hybrid data structure)
• Insertion order is not preserved and it is based onHashCode of keys	• Insertion order is preserved.
• Introduced in 1.2 version	• Introduced in 1.4 version

- In the above HashMap program if we replace HashMap with LinkedHashMap then output is {malachi=700, Renuka=800, issacraj=200, jeswica=500} that is insertion order is preserved

```

import java.util.*;
class LinkedHashMapDemo
{
    public static void main(String[] args)
    {
        LinkedHashMap m = new LinkedHashMap();
        m.put("malachi",700);
        m.put("Renuka",800);
        m.put("issacraj",200);
        m.put("jeswica",500);

        System.out.println(m);
        // {malachi=700, Renuka=800, issacraj=200, jeswica=500}
        System.out.println(m.put("malachi",1000)); //700

        Set s = m.keySet();
        System.out.println(s); // [malachi, Renuka, issacraj, jeswica]

        Collection c = m.values();
        System.out.println(c); // [1000, 800, 200, 500]

        Set s1 = m.entrySet();
        System.out.println(s1);
        // [malachi=1000, Renuka=800, issacraj=200, jeswica=500]

        Iterator itr = s1.iterator();
        while (itr.hasNext())
        {
            Map.Entry m1 = (Map.Entry)itr.next();
            System.out.println(m1.getKey()+"----"+m1.getValue());
            /* malachi----1000
             * Renuka----800
             * issacraj----200
             * jeswica----500 */

            if (m1.getKey().equals("malachi"))
            {
                m1.setValue(10000);
            }
        }
        System.out.println(m);
        // {malachi=10000, Renuka=800, issacraj=200, jeswica=500}
    }
}

```

Note:

LinkedHashSet and LinkedHashMap are commonly used for developing cache based application

Difference between “==” and “.equals ()”

In general “==” operator meant for reference comparison (address comparison) whereas “.equals()” method meant for content comparison.

```
Integer I1 = new Integer (10);
Integer I2 = new Integer (10);
System.out.println (I1==I2);
System.out.println (I1. equals(I2)); true
```

IdentityHashMap:

- It is exactly same has HashMaps including methods and constructor except the following difference
- In the case of normal HashMap JVM will use “.equals()” method to identify duplicate keys which is meant for context comparison.

But in the case of identity HashMap JVM will use “==” operator to identify duplicate keys which is meant for reference comparison(address comparison)

```
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); // {10= kalyan}
```

- I1 and I2 are duplicate keys because I1.equals(I2) returns true.
- If we are replace HashMap with identity HashMap the I1 and I2 are not duplicate keys, because I1 == I2 returns false.
- In this case output is {10 = pawan, 10 = kalyan}

WeakHashMap:

It is exactly same as HashMap except the following difference.

In this case of HashMap even though object doesn't have an reference it is not eligible for garbage collection if it is associated with HashMap that is HashMap dominates garbage collector, but in the case of WeakHashMap if object doesn't contain any reference it is eligible for garbage collector even though object associated with WeakHashMap that is garbage collector dominates WeakHashMap.

```

import java.util.*;
class WeakHashMapDemo
{
    public static void main(String[] args) throws Exception

    {
        HashMap m = new HashMap();
        Temp t = new Temp();
        m.put(t,"malachi");
        System.out.println(m);//{Temp=malachi}
        t = null;
        System.gc();
        Thread.sleep(5000);
        System.out.println(m);//{Temp=malachi}
    }
}

class Temp
{
    public String toString()
    {
        return "Temp";
    }
    public void finalize()
    {
        System.out.println("finalize method called");
    }
}

```

- In the above example Temp object not eligible for garbage collection because it is associated with HashMap in this case output is
{Temp=malachi}
{Temp=malachi}
- In the above program if we replace HashMap with WeakHashMap then Temp object eligible for garbage collection in this case output is
{Temp=malachi}
Finalize method called
{}

```

import java.util.*;
class WeakHashMapDemo1
{
    public static void main(String[] args) throws Exception

    {
        WeakHashMap m = new WeakHashMap();
        Temp t = new Temp();
        m.put(t,"malachi");
        System.out.println(m);//{Temp=malachi}
        t = null;
        System.gc();
    }
}

```

```

        Thread.sleep(5000);
        System.out.println(m); // {Temp=malachi}
    }
}

class Temp
{
    public String toString()
    {
        return "Temp";
    }

    public void finalize()
    {
        System.out.println("finalize method called");
    }
}

```

SortedMap:

- It is the child interface map
- If we want to represent a group of object as key value pair according to some sorting order of keys then we should o for sorted map.
- Sorting is based on the key but not based on value
- SortedMap defined the following specific methods
 - Object firstKey()
 - Object lastKey()
 - SortedMap headMap(Object key)
 - SortedMap tailMap(Object key)
 - SortedMap submap(Object key1, Object key2)
 - Comparator comparator()

Example:

Key	value
101	A
103	B
104	C
107	D
125	E
136	F

firstKey()	=> 101
lastKey()	=> 136
headMap(107)	=> {101 = A, 103 = B, 104 = C}
tailMap(107)	=> {107 = D, 125 = E, 136 = F}
submap(103,125)	=> {103 = B, 104 = C, 107 = D}
Comparator()	=> null

TreeMap:

- The underlaying data structure is REDBLACK Tree
- Insertion order is not preserved and it is based on some sorting order of keys
- Duplicate keys are not allowed but values can be duplicated.
- If we are depending on default natural sorting order then keys should be homogeneous and comparable otherwise, we will get runtime exception saying “ClassCastException” if we are defining our own sorting by comparator then keys need not be homogeneous and comparable, we can take heterogeneous non comparable Objects also.
- Whether we are depending on default natural sorting order or customized sorting order there are no restriction for values we can take heterogeneous non comparable objects also.

Null acceptance:

- For non-empty TreeMap, if we are trying to insert an entry with null key then we will get runtime exception saying “NullPointerException”.
- For empty TreeMap as the first entry with null key is allowed but after inserting that entry if we are trying to insert any other entry than we will get runtime exception saying “NullPointerException”.

Note:

The above Null acceptance rule applicable until 1.6 version. From 1.7 version onwards null is not allowed for “key”.

But for values we can use null any number of times there is no restriction whether it is 1.6 version or 1.7 version.

Constructors:

1. TreeMap t = new TreeMap()
For default natural sorting order
2. TreeMap t = new TreeMap(comparator c)
For customized sorting order
3. TreeMap t = new TreeMap(SortedMap m);
4. TreeMap t = new TreeMap(Map m);

Demo Program for default sorting order

```
import java.util.*;
class TreeMapDemo
{
    public static void main(String[] args)
    {
        TreeMap m = new TreeMap();
        m.put(100,"zzz");
        m.put(103,"YYY");
        m.put(101,"XXX");
```

```

        m.put(104,106);

        System.out.println(m);//{100=zzz, 101=XXX, 103=YYY, 104=106}

        m.put("fff","xxx");//java.lang.ClassCastException
        m.put(null,"xxx");//java.lang.NullPointerException
    }
}

```

Demo Program for Customized sorting order

```

import java.util.*;
class CustomizedSortingDemo
{
    public static void main(String[] args)
    {
        TreeMap t = new
                    TreeMap(new MyComparator());
        t.put("Malachi",10);
        t.put("Renuka",20);
        t.put("issacraj",30);
        t.put("jeswica",40);
        System.out.println(t);//{jeswica=40, issacraj=30, Renuka=20, Malachi=10}
    }
}
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    }
}

```

HashTable:

- The underlaying data structure for Hashtable is Hashtable.
- Insertion order is not preserved under it is based on HashCode of keys.
- Duplicate keys are not allowed and values can be duplicated
- Heterogeneous objects are allowed for both keys and values.
- Null is not allowed for both key and value otherwise we will get runtime exception saying “NullPointerException”.
- It implements serializable and cloneable interfaces but not Random Access.
- Every method present in Hashtable is synchronized and hence Hashtable object is thread safe, Hashtable is the best choice if our frequent operation is search operation.

constructor

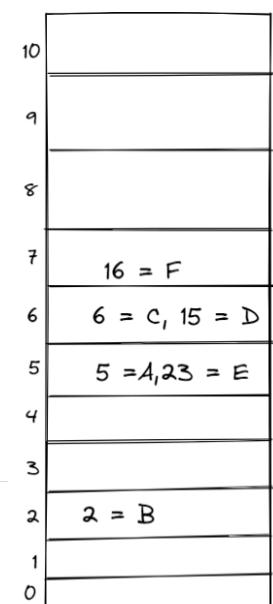
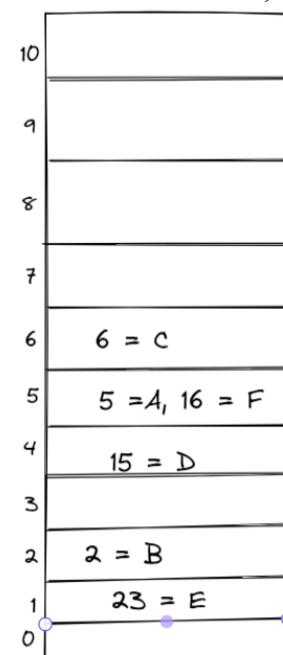
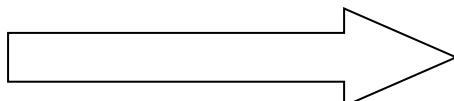
1. Hashtable h = new Hashtable();
Creates an empty Hashtable object with default InitialCapacity is 11 and default fill ration is 0.75.
2. Hashtable h = new Hashtable (int intialCapacity);
3. Hashtable h = new Hashtable (int intialCapacity float fillratio);
4. Hashtable h = new Hashtable (Map m);
Inter connection between Map objects

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(2),"B");  
        h.put(new Temp(6),"C");  
        h.put(new Temp(15),"D");  
        h.put(new Temp(23),"E");  
        h.put(new Temp(16),"F");  
        h.put("malachi",null);  
        System.out.println(h);  
    }  
}  
class Temp  
{  
    int i;  
    Temp(int i)  
    {  
        this.i = i;  
    }  
    public int hashCode()  
    {  
        return 1;  
    }  
    public String toString()  
    {  
        return i+"";  
    }  
}
```

- If we change hashCode method of Temp class

```
Public in hashCode()  
{  
    Return i % 9;  
}
```



24
23
...
...
...
16 16 = F
15 15 = D
6 6 = C,
5 5 = A
2 2 = B
0

7.

****VIP Properties:

- In our program if anything which changes frequently (like username, password, mailId, mobile No) are not recommended to hardcode in java program because if there is any change to reflect that change recompilation, rebuild and redeploy application are required even sometimes server restart also required which creates a big business impact to the client
- We can overcome this problem by using properties file such type of variable things we have to configure in the properties file form that properties file we have to read in to java program and we can use those properties.
- The main advantage of this approach is if there is a change in properties file to reflect that change just redeployment is enough which won't create any business impact to the client
- We can use java properties object to hold properties which are coming from properties file.
- In normal Map(like HashMap, Hashtable, TreeMap) key and value can be any type, but in the case of properties key and value should be string type

Constructor:

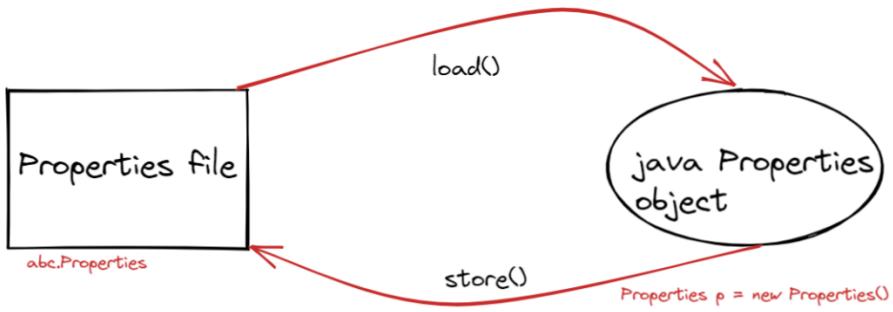
```
Properties p = new Properties();
```

Methods:

1. String setProperty (String pName, String pValue);
To set a new property, if the specified property all ready available the old value replaced new value and returns old value.
2. String getProperty (String pName);
To get value associated with the specified property if the specified property not available then this method returns "null".
3. Enumeration propertyNames (C);
Returns all property names present in properties object.
4. Void load (inputStream is);
To load properties from properties file into java properties object.

5. Void store (OutputStream os, String comment);

To store properties from java properties object into properties file.



```
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 s = p.getProperty("venki");
        System.out.println(s);

        p.setProperty("bhupathi","88888");
        FileOutputStream fos = new FileOutputStream("abc.properties");
        p.store(fos,"updated by malachi java notes");
    }
}
```

Output

```
In abc.properties file
#updated by malachi java notes
#Fri Nov 11 12:58:49 IST 2022
user=scott
venki=9999
bhupathi=88888
pwd=tiger
```

```
import java.util.*;
import java.io.*;
class PropertiesDemo2
{
    public static void main(String[] args) throws Exception
    {
        Properties p = new Properties();
        FileInputStream fis = new FileInputStream("db.properties");
        p.load(fis);
```

```

        String url = p.getProperty("url");
        String user = p.getProperty("user");
        String pwd= p.getProperty("pwd");
        Connection con = DriverManager.getConnection(url,user,pwd);
    }
}

```

Queue(I)

- 1.5 version enhancement (Queue(I));
- It is a child interface of collection.



- If we want to represent a group of individual objects according to prior to processing the we should go for queue.

For example:

Before sending SMS message all mobileNo, we have to store in some data structure. In which order we added mobileNo in the same order only message should be delivered for this FIRST IN FIRST OUT requirement queue is the best choice.

- Usually queue follows FIRST IN FIRST OUT order based on out requirement we can implement our own priority order also (priority queue) from 1.5 version onwards. LinkedList class also implements queue interface. LinkedList based implementations of queues follows FIRST IN FIRST OUT.

Queue interface specific methods:

1. Boolean offer (Object o)
To add an object into 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 head element of the queue. If queue is empty then this method raises Runtime exception saying “NoSuchElementException”
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 Runtime Exception saying “NoSuchElementException”.

PriorityQueue:

- If we want to represent a group of individual objects prior to processing according to some priority then we should go for priority queue.
- The priority can be either default natural sorting order or customized sorting order defined by comparator insertion order is not preserved and it is based on some priority.
- Duplicate objects are not allowed
- If we are depending on default natural sorting order. The compulsory object should be homogeneous and comparable otherwise will get Runtime Exception saying “ClassCastException”.
- If we are defining our own sorting by comparator the objects need not be homogeneous and comparable
- Null is not allowed even as the first element also

Constructor:

1. PriorityQueue q = new PriorityQueue ();
Creates any empty PriorityQueue with default InitialCapacity is 11 and all objects will be inserted according to default natural sorting order.
2. PriorityQueue q = new PriorityQueue (int initialCapacity);
3. PriorityQueue q = new PriorityQueue (int initialCapacity, Comparator c);
4. PriorityQueue q = new PriorityQueue (SortedSet s);
5. PriorityQueue q = new PriorityBlockingQueue (Collection c)

Example:

```
import java.util.*;
class PriorityQueueDemo
{
    public static void main(String[] args)
    {
        PriorityQueue q = new PriorityQueue();
        System.out.println(q.peek());//null
        //System.out.println(q.element());//java.util.NoSuchElementException
        for (int i = 0;i<=10 ;i++ )
        {
            q.offer(i);
        }
        System.out.println(q);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        System.out.println(q.poll());//0
        System.out.println(q);//[1, 3, 2, 7, 4, 5, 6, 10, 8, 9]
    }
}
```

Note:

Some platforms won't provide proper support for thread priority's and PriorityQueue.

Demo program for customized priority

```
import java.util.*;
class PriorityQueueDemo1
{
    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, B, L, A]
    }
}
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = (String)obj1;
        String s2 = (String)obj2;
        return s2.compareTo(s1);
    }
}
```

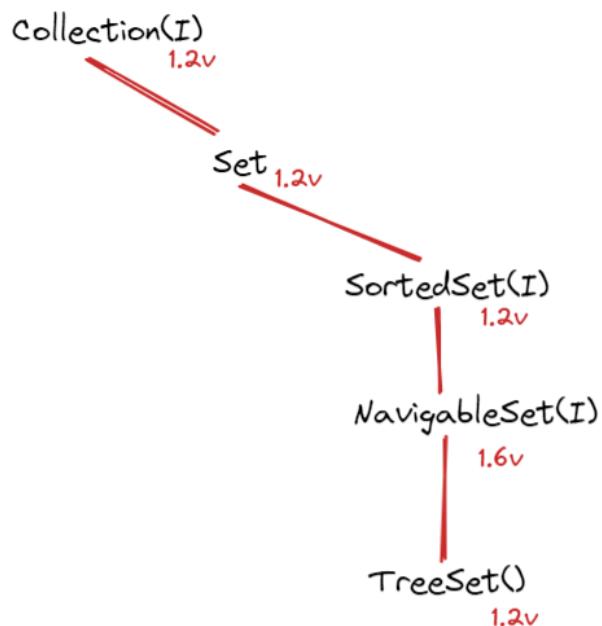
1.6 version enhancement in collection framework

Has a part of 1.6 version the following two concepts introduced in collection framework.

1. NavigableSet(I)
2. NavigableSet(I)

NavigableSet(I)

It is the child interface of SortedSet and it defines several methods for navigation purposes



NavigableSet define the following 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

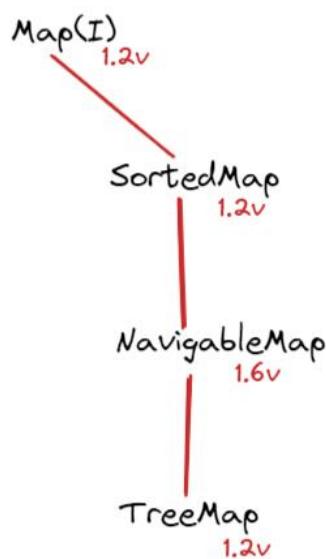
```

import java.util.*;
class NavigableSetDemo
{
    public static void main(String[] args)
    {
        TreeSet <Integer> t = new TreeSet<Integer>();
        t.add(1000);
        t.add(2000);
        t.add(3000);
        t.add(4000);
        t.add(5000);
        System.out.println(t);//[1000, 2000, 3000, 4000, 5000]
        System.out.println(t.ceiling(2000));//2000
        System.out.println(t.higher(2000));//3000
        System.out.println(t.floor(3000));//3000
        System.out.println(t.lower(3000));//2000
        System.out.println(t.pollFirst());//1000
        System.out.println(t.pollLast());//5000
        System.out.println(t.descendingSet());//[4000, 3000, 2000]
        System.out.println(t);//[2000, 3000, 4000]
    }
}

```

NavigableMap(I)

NavigableMap is child Interface of SortedMap it defines several methods for navigation purpose.



NavigableMap defines the following methods:

1. floorKey(e)
2. lowerKey(e)
3. ceilingKey(e)
4. higherKey(e)
5. pollFirstEntry()
6. pollLastEntry()
7. descendingMap()

```
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(t); // {a=apple, b=banana, c=cat, d=dog, g=gun}
        System.out.println(t.ceilingKey("c")); // c
        System.out.println(t.higherKey("a")); // b
        System.out.println(t.floorKey("d")); // d
        System.out.println(t.pollFirstEntry()); // a=apple
        System.out.println(t.pollLastEntry()); // g=gun
        System.out.println(t.descendingMap()); // {d=dog, c=cat, b=banana}
        System.out.println(t); // {b=banana, c=cat, d=dog}
    }
}
```

collections

Collections class defines several utility methods for collections objects like Sorting, Searching, Reversing etc.,

Sorting elements of List:

Collections class defines the following to sort methods.

1. Public static void sort(List l)

To sort based on default natural sorting order, in this case `l` should compulsory contains homogeneous and comparable objects otherwise we will get runtime exception saying “ClassCastException”.

List should not contain null otherwise we will get “NullPointerException”

2. Public static void sort(List l, comparator c)

To start base on customized sorting order

Demo program for Sorting elements of list according to default natural sorting order

```
import java.util.*;
class CollectionSortDemo
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add("L");
        l.add("A");
        l.add("K");
        l.add("N");
        //l.add(new Integer(10));//java.lang.ClassCastException:
        //l.add(null); //java.lang.NullPointerException
        System.out.println("before sorting:"+l);//before sorting:[L, A, K, N]
        Collections.sort(l);
        System.out.println("after sorting:"+l);//after sorting:[A, K, L, N]
    }
}
```

Demo program to sort elements of list a according to customize sorting

```
import java.util.*;
class CollectionSortDemo1
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add("L");
        l.add("A");
        l.add("K");
        l.add("N");
        //l.add(new Integer(10));//java.lang.ClassCastException:
        //l.add(null); //java.lang.NullPointerException
        System.out.println("before sorting:"+l);//before sorting:[L, A, K, N]
        Collections.sort(l);
        System.out.println("after sorting:"+l);//after sorting:[A, K, L, N]
        Collections.sort(l,new MyComparator());
    }
}
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = (String)obj1;
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    }
}
```

Searching elements of List

Collections class define the following “Binary Search” methods

1. Public static in binarySearch (List l, Object target);
If the list is sorted according to default natural sorting order the we have to use this method.
2. Public static in binarySearch (List l, Object target, Comparator c)
We have to use this method if the List is sorted according customize sorting order.

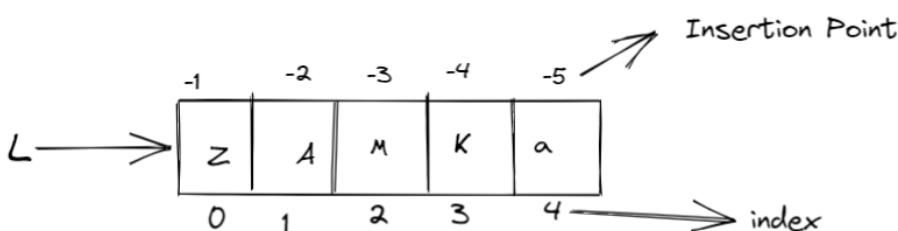
Conclusions:

- The above search methods internally will used binary search algorithm
- Successfully search return index unsuccessfully search returns insertion point.
- Insertion point is the location where we compare target element in the sorted list.
- Before calling binary search method compulsory list should be sorted otherwise we will get un-predictable results.
- ** if the list is sorted according to comparator the at the time of search operation also we have to pass same comparator object otherwise we will get un-predictable results.

```
import java.util.*;
class CollectionsSearchDemo
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add("Z");
        l.add("M");
        l.add("A");
        l.add("K");
        l.add("a");
        System.out.println(l);//[Z, M, A, K, a]

        Collections.sort(l);
        System.out.println(l);//[A, K, M, Z, a]

        System.out.println(Collections.binarySearch(l, "Z"));//3
        System.out.println(Collections.binarySearch(l, "J"));//-2
    }
}
```



Collection.binarySearch(l, "Z"); -> 3
Collection.binarySearch(l, "J"); -> 2

```

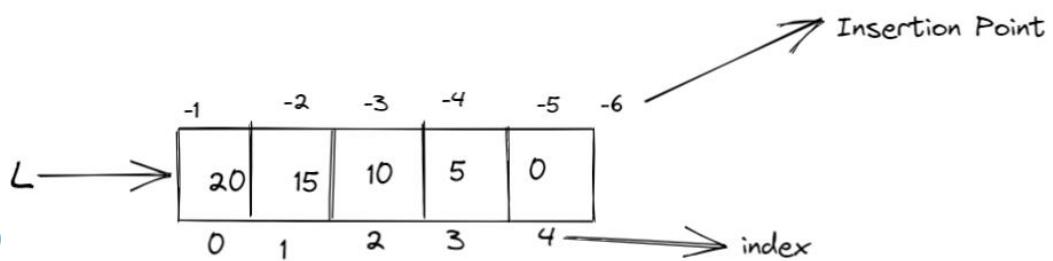
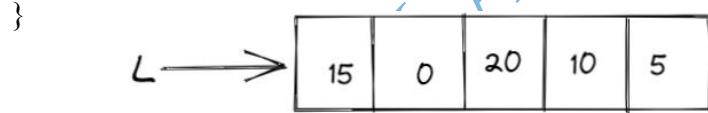
import java.util.*;
class CollectionsSearchDemo1
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add(15);
        l.add(0);
        l.add(20);
        l.add(10);
        l.add(5);
        System.out.println(l); // [15, 0, 20, 10, 5]

        Collections.sort(l, new MyComparator());
        System.out.println(l);

        System.out.println(Collections.binarySearch(l, 10, new MyComparator())); // 2
        System.out.println(Collections.binarySearch(l, 13, new MyComparator())); // -3
        System.out.println(Collections.binarySearch(l, 20)); // -6 unpredictable
    }
}

class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        Integer i1 = (Integer) obj1;
        Integer i2 = (Integer) obj2;
        return i2.compareTo(i1);
    }
}

```



```

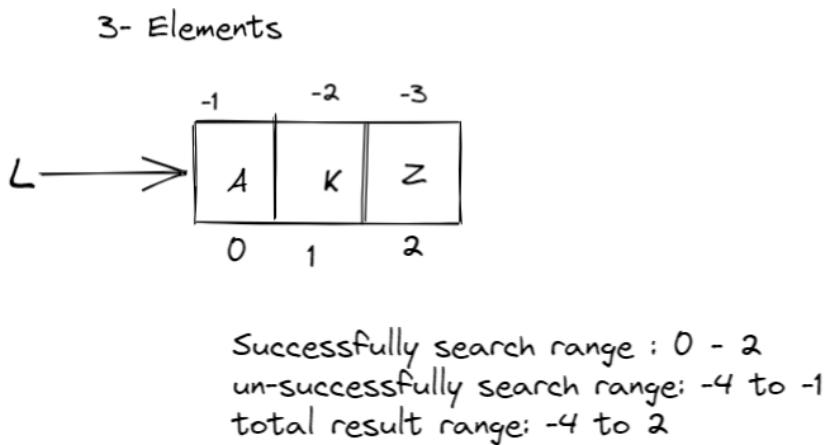
Collection.binarySearch(l, 10, new MyComparator()); -> 2
Collection.binarySearch(l, 13, new MyComparator()); -> -3
Collection.binarySearch(l, 17, new MyComparator()); -> unpredictable
Collection.binarySearch(l, 17, new MyComparator()); -> -2

```

Note:

1. For the List of n elements in the case of binary search method
 - a. Successfully search result range e = 0 to n-1
 - b. Un-successfully search result range is -(n+1)
 - c. Total result range is -(n+1) to n-1

Example:



Reversing elements of list:

Collections class defines the following reverse method to reverse elements of List.

Public static void reverse (List l)

```
import java.util.*;  
class CollectionReverseDemo  
{  
    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);  
        System.out.println(l);// [15, 0, 20, 10, 5]  
  
        Collections.reverse(l);  
        System.out.println(l);// [5, 10, 20, 0, 15]  
    }  
}
```

reverse () V/S reverseOrder ()

- we can use reverse method to reverse order of elements of list.
- Where as we can use reverse order method to get reversed Comparator.

Comparator c1 = Collections.reverseOrder (Comparator c);

C1 => descending order Comparator c => ascending order

Arrays

Arrays class is a utility class to define several utility methods for arrays (array object)

Sorting elements of array:

Arrays class defines the following sort methods to sort elements of primitives and object type arrays.

1. Public static void sort (Primitive[] p)
To sort according to natural sorting order.
2. Public static void sort (Object[] o);
To sort according to natural sorting order
3. Public static void sort (Object[] o, Comparator c);
To sort according to customized sorting order.

```
import java.util.Comparator;
import java.util.Arrays;
class ArraysSortDemo
{
    public static void main(String[] args)
    {
        int[] a = {10,5,20,11,6};
        System.out.println("primitive array before sorting");
        for(int a1 :a)
        {
            System.out.println(a1);
        }
        Arrays.sort(a);
        System.out.println("primitive array after sorting: ");
        for (int a1 :a )
        {
            System.out.println(a1);
        }

        String s[] = {"a","z","x","b","y"};
        System.out.println("primitive array before sorting");
        for(String a2 :s)
        {
            System.out.println(a2);
        }
        Arrays.sort(s);
        System.out.println("primitive array after sorting: ");
        for (String a2 :s )
        {
            System.out.println(a2);
        }

        Arrays.sort(s,new MyComparator());
        System.out.println("Object array after sorting by comparator");
    }
}
```

```
        for (String a3:s )
        {
            System.out.println(a3);
        }
    }
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    }
}
```

Output:

primitive array before sorting

10
5
20
11
6

primitive array after sorting:

5
6
10
11
20

primitive array before sorting

a
z
x
b
y

primitive array after sorting:

a
b
x
y
z

Object array after sorting by comparator

z
y
x
b
a

Note:

we can sort primitive arrays only based on default natural sorting order where as we can sort object arrays either based on default natural sorting ordered or based on customized sorting order.

Searching Elements of Array:

Arrays class defines the following binary search methods

1. Public static int binarySearch (primitive [] p, primitive target);
2. Public static int binarySearch (Object [] a, Object target)
3. Public static in binarySearch (Object [] a, Object target, Comparator c);

Note:

All rule of array class binary search method are exactly same as Collections class binary search methods.

```
import java.util.*;
import static java.util.Arrays.*;
class ArraysSearchDemo
{
    public static void main(String[] args)
    {
        int[] a = {10,5,20,11,6};
        Arrays.sort(a);
        System.out.println(Arrays.binarySearch(a,6));//1
        System.out.println(Arrays.binarySearch(a,11));//3
        System.out.println(Arrays.binarySearch(a,19));//-5

        String[] s = {"a","z","x","b","y"};
        Arrays.sort(s);
        System.out.println(binarySearch(s,"z")); //4
        System.out.println(binarySearch(s,"x")); //2
        System.out.println(binarySearch(s,"s")); //-3

        Arrays.sort(s,new MyComparator());
        System.out.println(binarySearch(s,"z", new MyComparator())); //0
        System.out.println(binarySearch(s,"x", new MyComparator())); //2
        System.out.println(binarySearch(s,"s", new MyComparator())); // -4
    }
}

class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    }
}
```

1.

10	5	20	11	6
----	---	----	----	---

-1	-2	-3	-4	-5	-6
5	6	10	11	20	
0	1	2	3	4	

Array.binarySearch(9,6); -> 1
 Array.binarySearch(9,14); -> -5

2.

A	Z	B
11	6	
-1	-2	-3

Array.binarySearch(s,"Z"); -> 2
 Array.binarySearch(s,"S"); -> -3

3.

A	Z	B
11	6	
-1	-2	-3

Z	B	A
0	1	2

binarySearch(s,"Z",new Comparator()); -> 0
 binarySearch(s,"S",new Comparator()); -> -2
 binarySearch(s,"n"); -> unpredictable result

Conversion of array to list

Public static List asList(Object a);

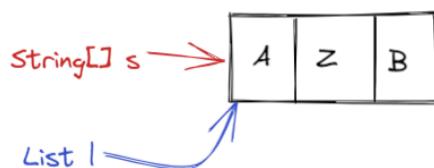
- Strictly speaking this method won't create an independent list object for the existing array we are getting list view.

Example:

```
String[] s = {"A", "Z", "B"}  

List l = Arrays.asList(s);
```

- By using Array reference, we perform any changes automatically that change will be reflected to the list similarly by using list reference if we perform any change that changes will be reflected automatically to the Array



- By using list reference we can't perform any operation which varies the size otherwise we will get runtime exception saying “unsupportedOperationException”.

```

l.add("M");
l.remove(l);
}

```

RuntimeException:
unsupported Operation Exception

But `l.set(l, "N")` → Perfectly valid

- By using list reference we are not allowed to replace with heterogeneous objects otherwise will get Runtime Exception saying “ArrayStoreException”.

```

l.set(l, new Integer(10));
RE: ArrayStoreException.

```

Example:

```

import java.util.*;
class ArrayAsListDemo
{
    public static void main(String[] args)
    {
        String[] s = {"A", "Z", "B"};
        List l = Arrays.asList(s);
        System.out.println(l); // [A, Z, B]

        s[0] = "K";
        System.out.println(l); // [K, Z, B]

        l.set(1,"L");
        for(String s1:s)
            System.out.print(s1+" "); // K L B

        l.add("malachi");
        System.out.println(l); // java.lang.UnsupportedOperationException

        l.remove(1);
        System.out.println(l); // java.lang.UnsupportedOperationException

        l.set(1,new Integer(10));
        System.out.println(l); // java.lang.ArrayStoreException
    }
}

```

Concurrent Collection

Need of Concurrent collections

- Tradition collection object(like ArrayList, HashMap ect) can be accessed by multiple threads simultaneously and there may be a chance of data inconsistency problems and hence these are not thread safe.
- Already existing thread safe collections(Vector, Hashtable, synchronizedList(), synchronizedSet(), synchronizedMap()) performance wise not up to mark.
- Because for every operations even for read operations also total collections will be loaded by only one thread at a time and it increases waiting time of threads.

Example:

```
import java.util.ArrayList;
import java.util.Iterator;
class ConcurrentCollectionsDemo
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add("A");
        l.add("B");
        l.add("C");
        Iterator itr = l.iterator();
        while(itr.hasNext())
        {
            String s = (String)itr.next();
            System.out.println(s);
        }
        l.add("D");//java.util.ConcurrentModificationException
    }
}
```

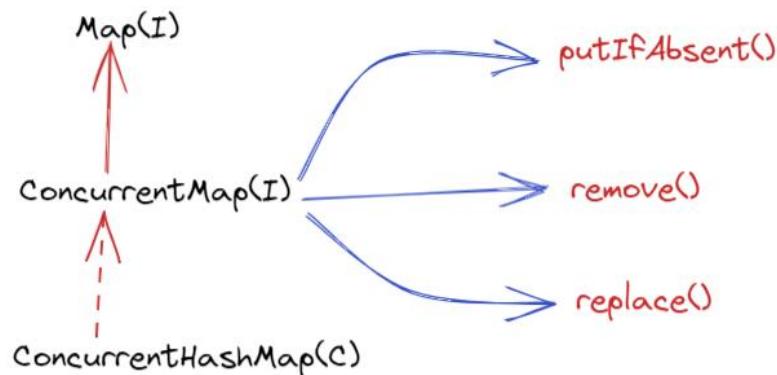
- Another big problem with traditional collections while one thread iterating collection, the other threads are not allowed to modify collections object simultaneously. If we are trying to modify then we will get concurrent modification exception.
- Hence these traditional collections objects are not suitable for scalable multi-threaded applications.
- To overcome these problems SUN people introduced concurrent collections in 1.5 version.

Difference between traditional and concurrent collections:

1. Concurrent collections are always thread safe.
2. When compared with traditional thread sage collections performance is more because of different looking mechanism.
3. While one thread interacting collection the other threads are allowed to modify collection in safe manner.
4. Hence concurrent collections never threw “ConcurrentModificationException”.
5. The important concurrent classes are:
 - a. ConcurrentHashMap
 - b. CopyOnWriteArrayList
 - c. CopyOnWriteArraySet.

ConcurrentHashMap(I)

ConcurrentMap(I)



Methods:

1. Object putIfAbsent(Object key, Object value)

To add entry to the map if the specified key is not already available

```
Object putIfAbsent(Object key, Object value)
If(!Map.containsKey(key))
{
    Map.put(key, value);
}
Else
{
    Return Map.get(key);
}
```

put(): if the key already available, old value will be replaced with new value and returns old value.

putIfAbsent (): if the key is already present the entry won't be added and returns old associated value. If the key is not available then only entry will be added.

```
import java.util.concurrent.ConcurrentHashMap;
class ConcurrentHashMapDemo
{
    public static void main(String[] args)
    {
        ConcurrentHashMap m = new ConcurrentHashMap();
        m.put(101, "malachi");
        m.put(101, "renuka");
        System.out.println(m); // {101=renuka}
        m.putIfAbsent(101, "bhupathi");
        System.out.println(m); // {101=renuka}
    }
}
```

2. boolean remove(Object key, Object value)

Removes the entry if the key associated with specified value only.

```
If(Map.containsKey(key) && Map.get(key).equals(value))
{
    Map.remove(key);
    Return true;
}
Else
{
    Return false;
}
```

Example:

```
import java.util.concurrent.ConcurrentHashMap;
class ConcurrentHashMapDemo1
{
    public static void main(String[] args)
    {
        ConcurrentHashMap m = new ConcurrentHashMap();
        m.put(101, "malachi");
        m.remove(101, "renuka"); // value not matched with key so not removed.
        System.out.println(m); // {101=malachi}
        m.remove(101, "malachi");
        System.out.println(m); // {}
    }
}
```

3. boolean replace(Object key, Object oldValue Object newValue)

boolean replace(Object key, Object oldValue, Object newValue)

if the key value matched
then replace with

```
If(Map.containsKey(key) && Map.get(key).equals(oldValue))
{
    Map.put(key, newValue);
    Return true;
}
Else
{
    Return false;
}
```

```
import java.util.concurrent.ConcurrentHashMap;
class ConcurrentHashMapDemo2
{
    public static void main(String[] args)
    {
        ConcurrentHashMap m = new ConcurrentHashMap();
        m.put(101, "malachi");
        m.replace(101, "renuka", "bhupathi");
        System.out.println(m); // {101=malachi}
        m.replace(101, "malachi", "renuka");
        System.out.println(m); // {101=renuka}
    }
}
```

ConcurrentHashMap

1. Underlying data structure is Hashtable.
2. ConcurrentHashMap allows concurrent read and thread safe update operations.
3. To perform read operations thread won't require any lock. But to perform update operation thread requires lock but it is the lock of only a particular part of map (Buckets level lock or segment lock)
4. Instead of whole map concurrent update achieved by internally dividing map into smaller portion which is defined by concurrency level.
5. The default concurrency level is 16.
6. That is ConcurrentHashMap allows simultaneously read operation and simultaneously 16 write(update) operations
7. Null is not allowed for both keys and values.
8. While one thread iterating the other thread can perform update operation and ConcurrentHashMap never throw "ConcurrentModificationException"

Constructors:

- **ConcurrentHashMap m = new ConcurrentHashMap();**
Creates an empty ConcurrentHashMap with default initial capacity 16 and default fill ratio is 0.75 and default concurrency level is 16.
- **ConcurrentHashMap m = new ConcurrentHashMap(int intialCapacity)**
- **ConcurrentHashMap m = new ConcurrentHashMap (int IntialCapacity, float fillratio)**
- **ConcurrentHashMap m = new ConcurrentHashMap(int intialCapacity, float fillRatio, int concurrencyLevel)**
- **ConcurrentMap m = new Concurrent(Map m);**

Example 1:

```
import java.util.concurrent.ConcurrentHashMap;
class ConcurrentHashMapDemo3
{
    public static void main(String[] args)
    {
        ConcurrentHashMap m = new ConcurrentHashMap();
        m.put(101, "A");
        m.put(102, "B");
        m.putIfAbsent(103,"C");
        m.putIfAbsent(101, "D");
        m.remove(101,"D");
        m.replace(102,"B","E");
        System.out.println(m);// {101=A, 102=E, 103=C}
    }
}
```

Example 2:

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.*;
class ConcurrentHashMapDemo4 extends Thread
{
    //static HashMap m = new HashMap(); //java.util.ConcurrentModificationException
    static ConcurrentHashMap m = new ConcurrentHashMap();
    public void run()
    {
        try
        {
            Thread.sleep(2000);
        }
        catch(InterruptedException e ){ }
        System.out.println("child trhead updating map");
        m.put(103,"c");
    }
}
```

```

public static void main(String[] args) throws InterruptedException
{
    m.put(101, "A");
    m.put(102, "B");
    ConcurrentHashMapDemo4 chd = new ConcurrentHashMapDemo4();
    chd.start();
    Set s = m.keySet();
    Iterator itr = s.iterator();
    while (itr.hasNext())
    {
        Integer i1 =(Integer) itr.next();
        System.out.println(" main thread itterating and current entry is : "
+ i1+"-----"+m.get(i1));
        Thread.sleep(3000);
    }
    System.out.println(m);
}
/*
output:
main thread itterating and current entry is : 101-----A
child tthread updating map
main thread itterating and current entry is : 102-----B
main thread itterating and current entry is : 103-----c
{101=A, 102=B, 103=c}
*/

```

- Update and we won't get any "ConcurrentModificationException".
- If we replace ConcurrentHashMap with HashMap the we will get "ConcurrentModificationException".

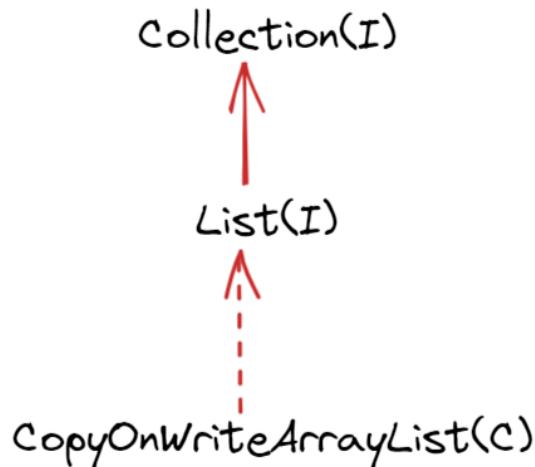
Difference between HashMap and ConcurrentHashMap

HashMap	ConcurrentHashMap
• It is not thread safe	• It is thread safe
• Relatively performance is high because threads are not required to wait to operate on HashMap.	• Relatively performance is low because some times threads are required to wait to operate a ConcurrentHashMap.
• While one thread iterating HashMap the other threads are not allowed to modify Map objects otherwise we will get Runtime Exception saying "ConcurrentModificationException".	• While on thread interacting ConcurrentHashMap the other threads are allowed to modify map objects in safe manner and it won't throw "ConcurrentModificationException"
• Iterator of HashMap is fail-fast and it throws ConcurrentModificationException.	• Iterator of ConcurrentHashMap is fail-safe and it won't throws ConcurrentModificationException
• Null is allowed for both keys and values	• Null is not allowed for both keys and values, otherwise we will get NullPointerException
• Introduced in 1.2 version	• Introduced in 1.5 version

Difference between ConcurrentHashMap, synchronizedMap () and Hashtable

ConcurrentHashMap	synchronizedMap ()	Hashtable
✓ We will get thread safety without locking total Map object just with Bucket level lock	✓ We will get thread safety by locking whole Map object	✓ We will get thread safety by locking whole Map object
✓ At-a-time multiple threads are allowed to operate on Map objects in safe manner	✓ At-a-time only one thread is allowed to perform any operations on Map object	✓ At-a-time only one thread is allowed to operate on Map object
✓ Read operations can be performed without lock but write operation can be performed with Bucket level lock	✓ Every read and write operations require total map object lock	✓ Every read and write operations require total Map object lock
✓ While one thread iterating Map object. The other threads are allowed to modify map and we won't get ConcurrentModificationException	✓ While one thread iterating map object the other threads are not allowed to modify map. Otherwise we will get ConcurrentModificationException	✓ While one thread iterating map object, the other thread is not allowed to modify map otherwise we will get ConcurrentModificationException
✓ Iterator of ConcurrentHashMap is "fail-safe" and won't raise "ConcurrentModificationException"	✓ Iterator of synchronizedMap () is "fail-fast" and it will raise "ConcurrentModificationException"	✓ Iterator of synchronizedMap is "fail-fast" and it will raise "ConcurrentModificationException"
✓ Null is not allowed for Both keys and values	✓ Null is allowed for both keys and values	✓ Null is not allowed for both keys and values.
✓ Introduced in 1.5 version	✓ Introduced in 1.2 version	✓ Introduced in 1.0 version

WriteArrayList(C)



- ✓ It is a thread safe version of `ArrayList` as the name indicates “`CopyOnWriteArrayList` creates a cloned copy of underlying “`ArrayList`” for every update operation at certain point both will synchronize automatically which is taken care by JVM internally.
- ✓ As update operation will be performed on cloned copy there is no effect for the threads which performs read operations.
- ✓ It is costly to use because for every update operation a cloned copy will be created. Hence copy on write `ArrayList` is the best choice of several read operations and less number of write operations are required to perform .
- ✓ Insertion order is preserved.
- ✓ Duplicate objects are allowed.
- ✓ Heterogeneous objects are allowed.
- ✓ Null insertion is possible
- ✓ It implements `Serializable`, `Cloneable` and `RandomAccess` interfaces.
- ✓ While threads are iterating are allowed to modify and we won't get `ConcurrentModificationException` that is iterator is fail-safe.
- ✓ Iterator of `ArrayList` can perform remove operation but iterator of copy on write `ArrayList` can't perform remove operation, otherwise we will get runtime Exception saying “`UnsupportedOperationException`”.

Constructors:

1. `CopyOnWriteArrayList l = new CopyOnWriteArrayList();`
2. `CopyOnWriteArrayList l = new CopyOnWriteArrayList(Collection c);`
3. `CopyOnWriteArrayList l = new CopyOnWriteArrayList(Object[] a);`

Methods:

1. Boolean addIfAbsent(Object o):

The element will be added if and only if list doesn't contain this element.

Example:

```
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.ArrayList;
class CopyOnWriteArrayListDemo1
{
    public static void main(String[] args)
    {
        CopyOnWriteArrayList l = new CopyOnWriteArrayList();
        l.add("A");
        l.add("A");
        l.addIfAbsent("B");
        l.addIfAbsent("B");
        System.out.println(l); // [A, A, B]
    }
}
```

2. Int addAllAbsent(Collection c)

The elements of collections will be added to the list if elements are absent and returns number of elements added.

Example 1:

```
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.ArrayList;
class CopyOnWriteArrayListDemo {
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add("A");
        l.add("B");

        CopyOnWriteArrayList L1 = new CopyOnWriteArrayList();
        L1.add("A");
        L1.add("C");
        System.out.println(L1); // [A, C]
        L1.addAll(l);
        System.out.println(L1); // [A, C, A, B]

        ArrayList L2 = new ArrayList();
        L2.add("A");
        L2.add("D");
        L1.addAllAbsent(L2);
        System.out.println(L1); // [A, C, A, B, D]

    }
}
```

Example 2:

```
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.*;
class CopyOnWriteArrayListDemo2 extends Thread
{
    static CopyOnWriteArrayList l = new CopyOnWriteArrayList();
    public void run()
    {
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException e )
        {
        }
        System.out.println("child thread updating list");
        l.add("C");
    }
    public static void main(String[] args) throws InterruptedException
    {
        l.add("A");
        l.add("B");
        CopyOnWriteArrayListDemo2 cwl = new CopyOnWriteArrayListDemo2();
        cwl.start();
        Iterator itr = l.iterator();
        while (itr.hasNext())
        {
            String s = (String)itr.next();
            System.out.println("main thread iterating list and current object is: "+s);
            Thread.sleep(3000);
        }
        System.out.println(l);
    }
}
/*
output:
main thread iterating list and current object is: A
child thread updating list
main thread iterating list and current object is: B
[A, B, C]
*/
```

- ✓ In the above example while main thread iterating list child thread is allowed to modify and we won't get any ConcurrentModificationException.
- ✓ If we replace CopyOnWriteArrayList with ArrayList the we will get ConcurrentModificationException.
- ✓ Iterator of CopyOnWriteArrayList can't perform remove operation. Otherwise we will get runtime exception saying "UnsupportedOperationException".

```

import java.util.concurrent.CopyOnWriteArrayList;
import java.util.*;
class CopyOnWriteArrayListDemo3
{
    public static void main(String[] args)
    {
        CopyOnWriteArrayList l = new CopyOnWriteArrayList();
        l.add("A");
        l.add("B");
        l.add("C");
        l.add("D");
        System.out.println(l); // [A, B, C, D]
        Iterator itr = l.iterator();
        while (itr.hasNext())
        {
            String s = (String)itr.next();
            if (s.equals("D"))
            {
                itr.remove();
            }
        }
        System.out.println(l); // java.lang.UnsupportedOperationException
    }
}

```

- If we replace CopyOnWriteArrayList with ArrayList we won't get any unsupported Operation Exception.
- In this case the out put is
[A, B, C, D]
[A, B, C]

```

import java.util.concurrent.CopyOnWriteArrayList;
import java.util.*;
class CopyOnWriteArrayListDemo4
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add("A");
        l.add("B");
        l.add("C");
        l.add("D");
        System.out.println(l); // [A, B, C, D]
        Iterator itr = l.iterator();
        while (itr.hasNext())
        {
            String s = (String)itr.next();
            if (s.equals("D"))
            {
                itr.remove();
            }
        }
    }
}

```

```

        }
    }
    System.out.println(l); // [A, B, C]
}
}

```

Example:

```

import java.util.concurrent.CopyOnWriteArrayList;
import java.util.*;
class CopyOnWriteArrayListDemo5
{
    public static void main(String[] args)
    {
        CopyOnWriteArrayList l = new CopyOnWriteArrayList();
        l.add("A");
        l.add("B");
        l.add("C");
        Iterator itr = l.iterator();
        l.add("D");
        while (itr.hasNext())
        {
            String s = (String)itr.next();
            System.out.println(s);
        }
    }
}
/*
Output:
A
B
C
*/

```

Reason:

- Every update operation will be performed a separate copy. Hence after getting iterator if we are trying to perform any modification to the list it won't be reflected to the iterator.
- In the above program if we replace CopyOnWriteArrayList with ArrayList the we will get Runtime Exception saying java.util.ConcurrentModificationException

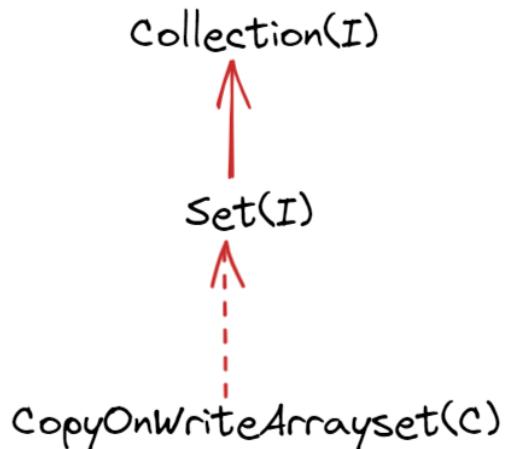
Difference between ArrayList and CopyOnWriteArrayList

ArrayList	CopyOnWriteArrayList
<ul style="list-style-type: none"> • It is not thread safe 	<ul style="list-style-type: none"> • It is thread safe because every update operation will be performed on separate cloned copy.
<ul style="list-style-type: none"> • While one thread iterating list object, the other threads are not allowed to modify list otherwise we will get ConcurrentModificationException 	<ul style="list-style-type: none"> • While one thread iterating list object, the other threads are allowed to modify list in safe manner and won't get ConcurrentModificationException
<ul style="list-style-type: none"> • Iterator is fail-fast 	<ul style="list-style-type: none"> • Iterator fail-safe
<ul style="list-style-type: none"> • Iterator of ArrayList can perform remove operation 	<ul style="list-style-type: none"> • Iterator of CopyOnWriteArrayList can't perform remove operations otherwise we will get runtime exception saying "UnsupportedOperationException".
<ul style="list-style-type: none"> • Introduced in 1.2 version present in java.util package 	<ul style="list-style-type: none"> • Introduced in 1.5 version present in java.util.concurrent package.

Difference between CopyOnWriteArrayList, synchronizedList (), vector(c)

CopyOnWriteArrayList ()	synchronizedList ()	Vector ()
<ul style="list-style-type: none"> • We will get thread safety because every update operations will be performed on separate cloned copy. 	<ul style="list-style-type: none"> • We will get thread safety because At-a-time list can be accessed by only one thread at-a-time. 	<ul style="list-style-type: none"> • We will get thread safety because at a time only one thread is allowed to access vector object.
<ul style="list-style-type: none"> • At a time multiple threads are allowed to access. Operate on CopyOnWriteArrayList 	<ul style="list-style-type: none"> • At a time only one thread is allowed to perform any operation on list object 	<ul style="list-style-type: none"> • At a time only one thread is allowed to operate on vector object.
<ul style="list-style-type: none"> • While one thread iterating list object, the other threads are allowed to modify and we won't get ConcurrentModificationException 	<ul style="list-style-type: none"> • While one thread iterating, the other threads are not allowed to modify list, otherwise we will get ConcurrentModificationException 	<ul style="list-style-type: none"> • While one thread iterating, the other threads are not allowed to modify vector, otherwise we will get ConcurrentModificationException
<ul style="list-style-type: none"> • Iterator is fail-safe and won't raise ConcurrentModificationException 	<ul style="list-style-type: none"> • Iterator is fail-fast and it will raise ConcurrentModificationException 	<ul style="list-style-type: none"> • Iterator is fail-fast and it will raise ConcurrentModificationException
<ul style="list-style-type: none"> • Iterator can't perform remove operation otherwise we will get UnsupportedOperationException 	<ul style="list-style-type: none"> • Iterator can perform remove operation 	<ul style="list-style-type: none"> • Iterator can perform Remove operation.
<ul style="list-style-type: none"> • Introduced in 1.5 version 	<ul style="list-style-type: none"> • Introduced in 1.2 version 	<ul style="list-style-type: none"> • Introduced in 1.0 version

CopyOnWriteArraySet



1. It is a thread safe version of set.
2. Internally implement by `CopyOnWriteArrayList`.
3. Insertion order is preserved.
4. Duplicate objects are not allowed.
5. Multiple threads can able to perform read operation simultaneously but for every update operation a separate cloned copy will be created.
6. As of every update operation a separate cloned copy will be created which is costly hence if multiple update operation are required then it is not recommended to use `CopyOnWriteArraySet`.
7. While one thread iterator set the other threads are allowed to modify set and we won't get `ConcurrentModificationException`
8. Iterator of `CopyOnWriteArraySet` can perform only read operation and won't perform remove operation other wise we will get runtime Exception saying "`UnsupportedOperationException`".

Constructor

1. `CopyOnWriteArraySet s = new CopyOnWriteArraySet();`
Creates an empty `CopyOnWriteArraySet` Object.
2. `CopyOnWriteArraySet s = new CopyOnWriteArraySet(Collection c);`
Creates `CopyOnWriteArraySet` Object which is equivalent to given collection Object.

```
import java.util.concurrent.CopyOnWriteArraySet;
import java.util.*;
class CopyOnWriteArraySetDemo
{
    public static void main(String[] args)
    {
        CopyOnWriteArraySet l = new CopyOnWriteArraySet();
        l.add("A");
        l.add("B");
    }
}
```

```

        l.add("C");
        l.add("D");
        l.add(null);
        l.add(10);
        l.add("D");
        System.out.println(l); // [A, B, C, D, null, 10]
    }
}

```

Methods:

Whatever methods present in collections and set interface are the only methods applicable for CopyOnWriteArrayList() and there are no special methods.

Difference between CopyOnWriteArrayList () and synchronizedList ()

CopyOnWriteArrayList ()	synchronizedList ()
<ul style="list-style-type: none"> It is thread safe because every update operation will be performed on separate cloned copy 	<ul style="list-style-type: none"> It is thread safe because at a time only one thread can perform operation.
<ul style="list-style-type: none"> While one thread iterating set, the other threads are allowed to modify and we won't get ConcurrentModificationException 	<ul style="list-style-type: none"> While one thread iterating the other threads are not allowed to modify set otherwise we will get ConcurrentModificationException
<ul style="list-style-type: none"> Iterator is fail-safe 	<ul style="list-style-type: none"> Iterator is fail-first
<ul style="list-style-type: none"> Iterator can perform only read operation and can't perform remove operation otherwise we will get runtime exception saying "unsupportedOperationException" 	<ul style="list-style-type: none"> Iterator can perform both read and remove operation.
<ul style="list-style-type: none"> Introduced in 1.5 version 	<ul style="list-style-type: none"> Introduced in 1.2 version



Generics

1. Introduction
2. Generic classes
3. Bounded types
4. Generic methods & wild card character(?)
5. Communication with non- generic code.
6. Conclusions

- The main objectives of generics are to provide type safety and to resolve type casting problems.

Case I

Type safety

Arrays are type safe that is we can give the guarantee for the type of elements present inside array.

For Example: if program 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 compile time error

```
String [] s = new String[1000];
S[0] = "malachi";
S[1] = "Renuka";
S[2] = new Integer(10); // incompatible types; found: java.lang.Integer required:
java.lang.String
S[2] = "bhupathi";
```

Hence String array can contain only String type of objects due to this we can give the guarantee for type of elements present inside array hence arrays are safe to use with respect to type. That is arrays are type safe.

But collections are not type safe that is we cannot give the guarantee for the type of elements present inside collections.

For example: if our program requirement is to hold only String type of objects, and we choose ArrayList, by mistake if we are trying to add any other type of object we won't get any compile time error but the program may fail at runtime.

```
//import java.util. concurrent.CopyOnWriteArraySet;
import java.util.*;
class CopyOnWriteArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add("malachi");
```

```

        l.add("Bhupathi");
        l.add("renuka");
        l.add(new Integer(10));
        String name0 = (String)l.get(0); //malachi
        String name1 = (String)l.get(1); //Bhupathi
        String name2 = (String)l.get(2); //renuka
        String name3 = (String)l.get(3); //java.lang.ClassCastException
        System.out.println(name0); //malachi
        System.out.println(name1); //bhupathi
        System.out.println(name2); //renuka
        System.out.println(name3); //java.lang.ClassCastException

    }
}

```

Hence, we cannot give the guarantee for the type of elements present in inside collection due to these collections are not safe to use with respect to type that is collections are not type safe.

Case 2 TypeCasting

In this case of arrays at time of retrieval it is not required to perform type casting because there is a guarantee for the type of elements present inside array.

```

class TypeCastingDemo
{
    public static void main(String[] args)
    {
        String[] s = new String[1000];
        s[0] = "malachi";
        s[1] = "renuka";
        s[2] = "bhupathi";
        s[3] = "jeswica";
        String name1 = s[0];
        String name2 = s[1];
        String name3 = s[2];
        String name4 = s[3];
        System.out.println(name1); //malachi
        System.out.println(name2); //renuka
        System.out.println(name3); //bhupathi
        System.out.println(name4); //jeswica
    }
}// type casting not required

```

But in the case of collections at the time of retrieval compulsory we should perform type casting because there is no guarantee for the type elements present inside collection.

```
import java.util.ArrayList;
```

```

class TypeCastingDemo
{
    public static void main(String[] args)
    {
        ArrayList s = new ArrayList();
        s.add("malachi");
        s.add("renuka");
        s.add("bhupathi");
        s.add("jeswica");
        String name1 = (String)s.get(0);
        String name2 = (String)s.get(1);
        String name3 = (String)s.get(2);
        String name4 = s.get(3); // CE: incompatible types found: java.lang.Object;
        required: java.lang.String
        System.out.println(name1); // malachi
        System.out.println(name2); // renuka
        System.out.println(name3); // bhupathi
        System.out.println(name4); // CE: incompatible types found: java.lang.Object;
        required: java.lang.String
    }
}

```

Hence type casting is a bigger headache in collections.

To overcome above problems of collections people introduced Generics concept in 1.5 version. Hence the main objective of generics are

1. To provide type safety.
2. To resolve type casting problem.

For example: to hold only String type of object we can create generic version of array list object as follows.

```
ArrayList<String> l = new ArrayList<String>();
```

For this ArrayList we can add only String type of objects by mistake if we are try to add any other type then we will get compile time error.

```

l.add("malachi");
l.add("bhupathi");
l.add(new Integer(10)); => Compile time error
l.add("Renuka");

```

hence through generic we are getting type safety.

At the time retrieval we are not required to perform type casting.

```
ArrayList<String> l = new ArrayList<String>();
l.add("malachi");
String name1 = l.get(0); // type casting Is not required
```

Hence, through generics we can solve type casting problem.

Difference between **ArrayList l= new ArrayList ()** and **ArrayList<String> l = new ArrayList<String> ()**

ArrayList l= new ArrayList ()	ArrayList<String> l = new ArrayList<String> ()
<ul style="list-style-type: none">• It is a non-generic version of ArrayList object	<ul style="list-style-type: none">• It is a generic version of ArrayList Object
<ul style="list-style-type: none">• It is for these ArrayList we can add any type of object and hence it is not type safe	<ul style="list-style-type: none">• For this array list we can add only String type of object and hence it is a type safe.
<ul style="list-style-type: none">• At the time of retrieval compulsory(type casting is required) we have to perform typecasting	<ul style="list-style-type: none">• At the time of retrieval we are not required to perform type casting.

Conclusion 1

Polymorphism concept applicable for only for the base type but not for parameter type (usage of parent reference to hold child object is the concept polymorphism)

```
ArrayList <String> l = new ArrayList<String>();  
ArrayList: base type; <String> : parameter Type;  
List<String> l = new ArrayList<String>();  
Collection<String> l = new ArrayList<String>();  
ArrayList<Object> l = new ArrayList<String>(); // CE: Incompatible types found:  
ArrayList<String> required: ArrayList(Object);
```

Conclusion 2

For the type parameter we can provide any class or interface name but not primitives we are trying to provide primitive the we will get compile time error.

```
ArrayList<int> x = new ArrayList<int>();  
// CE: unexpected type found: int required: reference.
```

Generic classes

Until 1.4 version a non-generic version ArrayList class declared as follows.

```
Class ArrayList
{
    Add (Object o);
    Object get(int index);
}
```

The argument o add method is object and hence we can add any type of object to the a ArrayList due to this we are missing type safety.

The return type of get method is Object hence at the time of retrieval ot perform type casting.

But in 1.5 version a generic version of ArrayList class is declared as follows

```
Class ArrayList<T>
{
    Add(T t);
    t.get(int index);
}
<T> : type parameter
```

Base on our runtime requirement T replaced with the 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> l= new ArrayList();
```

For this requirement compiler consider version of ArrayList class is 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 of Objects by mistake we are trying to add any other type we can get compile time error.

```
l.add("malachi");
l.add("new Integer(10)); // CE: cannot find symbol; symbol: method
add(java.lang.Integer); location: class ArrayList<String>
```

hence through generics we are getting type safety the return type of get() method is String and hence at the time of retrieval we are not require to perform type Casting.

```
String name1 = l.get(0); // type casting is not required .
```

In generics we are associating a type parameter to the class such type parameterized classes are nothing but generic classes are template classes.

Based on our requirement we can define our own generic class also.

Example:

```
Class Account<T>
{
}
Account<gold> a1 = new Account<gold>();
Account<platinum>a2 = new Account<platinum>();

class Gen<T>
{
    T ob;
    Gen(T ob)
    {
        this.ob = ob;
    }
    public void show()
    {
        System.out.println("the type of ob :" +ob.getClass().getName());
    }
    public T getob()
    {
        return ob;
    }
}
class GenericDemo
{
    public static void main(String[] args)
    {
        Gen<String> g1 = new Gen<String>("malachi");
        g1.show(); //the type of ob :java.lang.String
        System.out.println(g1.getob());//malachi

        Gen<Integer> g2 = new Gen<Integer>(10);
        g2.show(); //the type of ob :java.lang.Integer
        System.out.println(g2.getob());//10

        Gen<Double> g3 = new Gen<Double>(10.5);
        g3.show(); //the type of ob :java.lang.Double
        System.out.println(g3.getob());//10.5
    }
}
```

Bounded Types

We can bound the type parameter for a particular range by using extends keyword such types are called bounded types

```
Class Test<T>
{
}
```

As the type parameters we can pass any type and they are no restrictions and hence it is unbounded type.

```
Test<Integer> t1 = new Test<Integer> ();
Test<Integer> t2 = new Test<Integer> ();
```

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.

Example:

```
Class Test< T extends Number>
{
}
Test<Integer> T1 = new Test<Integer> ();
Test<String> T2 = new Test<String> ();
//CE: type parameter java.lang.String is not within bound
```

Example 2:

```
Class Test< T extends Runnable>
{
}
Test<Runnable> T1 = new Test<Runnable> ();
Test<Thread> T2 = new Test<Thread> ();
Test<Integer> T3 = new Test<Integer> ();
//CE: type parameter java.lang.Integer is not within its Bound
```

We can define bounded types even in combination also.

Example

```
Class Test<T extends number & Runnable>
```

As the type parameter we can take any thing which should be child class of number and should be implements runnable interface.

```
Class Test <T extends Runnable & Comparable>
```

```
Class Test< T extends Number & Runnable & Comparable>
```

```
Class Test<T extends Runnable & Number>
```

// Error: Because we have to take class first followed by interface next

```
Class Test<T extends Number & Thread>
```

// error: because we can't extend more than one class simultaneously.

Conclusions:

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 keywords purpose with extends keyword.

```
Class Test<T extends Number> => valid
```

```
Class Test<T implements Runnable> => invalid
```

```
Class Test<T extends Runnable> => valid
```

```
Class Test<T super String> => invalid
```

2. As the type parameter "T" we can take any valid java identifier but it is convention to use "T".

```
Class Test<T> => valid
```

```
Class Test<X> => valid
```

```
Class Test<A> => valid
```

```
Class Test<malachi> => valid
```

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>{ } => valid
```

```
Class Test<x, y, z>{ } => valid
```

```
Class HashMap<K, V>{ } => valid
```

```
HashMap<Integer, String> h = HashMap<Integer, String>(); => valid
```

Generic Methods and wild-card character(?)

1. M1(ArrayList<String> l)

We can call this method by passing ArrayList of only string type.

But within the method, we can add only String type of Object to the list. By mistake if we are trying to add any other type then we will get compile time error.

```
M1(ArrayList<String> l);
l.add("A"); => valid
l.add("Null"); => valid
l.add(10); => CE
```

2. M1(ArrayList<?> l)

We can call this method by passing ArrayList of any type

But within the method, we cannot add any thing to the list except null because we don't know the type exactly.

Null is allowed because it is valid value for any type

```
M1(ArrayList<?> l)
{
    l.add(10.5); => in-valid
    l.add("A"); => in-valid
    l.add(10); => in-valid
    l.add(null); => valid
}
```

This type of method best suitable for read only operations.

3. M1(ArrayList<? Extends X> l)

X can be either class or interface if X is a class then we can call this method by passing ArrayList of either X type are its child classes.

If X is an interface then we can call this method by passing ArrayList of either X types or its implementation classes.

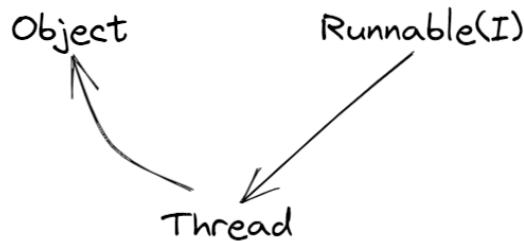
But with in the method we cannot add any thing to the list except null because we don't know the type exactly

This type of methods also best suitable for read only operations

4. M1(ArrayList<? Super X> l);

X can be either class or interface. If X is class then we can call this method by passing ArrayList of either "X" type or its super classes.

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 within the method we can add "X" type of objects and null to the list.

```
ArrayList<String> l = new ArrayList<String>(); => valid  
ArrayList<?> l = new ArrayList<String>(); => valid  
ArrayList<?> l = new ArrayList<Integer>(); => valid  
ArrayList<? Extends Number> l = new ArrayList<Integer>(); => valid  
ArrayList<? Extends Number> l = new ArrayList<String>(); => invalid  
                                         // CE: incompatible types; found: ArrayList<String>; Required:  
ArrayList<? Extends number>  
ArrayList<? Super String> l = new ArrayList<Object>(); => valid  
ArrayList<?> l = new ArrayList<?>(); => invalid  
                                         // CE: unexcepted type; found: ? required: class or interface without  
bounds.  
  
ArrayList<?> l = new ArrayList<? Extends Number>(); => invalid  
                                         // CE: unexcepted type; found: ? required: class or interface without  
bounds.
```

We can declare type parameter either at class level or at method level.

Declaring type Parameter at class Level

We can declare type parameter either at class level or at method level.

```
Class Test<T>  
{  
    We can use "T" with in this class based on our requirement at class level  
}
```

Declaring type parameter at method level

We have to declare type parameter just before return type

```
Class Test  
{  
    Public <T> void m1(T ob)  
    {  
        We can use "T" any where within this method based on our requirement.  
    }  
}
```

We can define bounded types even at method level also

```
Public <T> void m1()
    <T extends Number> => valid
    <T extends Runnable> => valid
    <T extends Number & Runnable> => valid
    <T extends Comparable & Runnable> => valid
    <T extends Number & comparable & Runnable> => valid
    <T extends Runnable & Number> => in-valid
        First we have to take class and the interface
    <T extends Number & Threads>
        We can't extend more than one class
```

Communication with Non-generic code

If we send generic object to non-generic area the it starts behaving like non-generic object similarly if we send non-generic object to generic area then it starts behaving like generic object that is the location in which object present based on that behaviour will be defined.

```
import java.util.*;
class NonGenericDemo
{
    public static void main(String[] args)
    {
        ArrayList<String> l = new ArrayList<String> ();
        l.add("malach");
        l.add("renuka");
        //l.add(10); //no suitable method found for add(int)
        m1(l);
        System.out.println(l);
    }
    public static void m1(ArrayList l)
    {
        l.add(10);
        l.add(10.5);
        l.add(true);
    }
}
```

Conclusions:

1. The main purpose of generics is to provide type safety and to resolve type casting problems.
2. Type safety and type casting both are applicable at compile time. Hence generic concepts applicable only at compile time but not at runtime at the time of compilation as a last step generic syntax will be removed and hence for the JVM generics syntax won't be available hence the following declarations are equal.

```
ArrayList l = new ArrayList<String>();  
ArrayList l = new ArrayList<Integer>();  
ArrayList l = new ArrayList<Double>();  
ArrayList l = new ArrayList();
```

Example:

```
Import java.util.*;  
Class ConclusionDemo  
{  
    Public static void main(String[] args)  
    {  
        ArrayList l = new ArrayList();  
        l.add(10);  
        l.add(10.5);  
        l.add(true);  
        System.out.println(l); // [10,10.5,true]  
    }  
}
```

3. The following declarations are equal

```
ArrayList<String> l = new ArrayList<String>();
```

```
ArrayList<String> l = new ArrayList<>()
```

For these ArrayList object we can add only String type of objects

Example:

```
Class Test  
{  
    Public void m1(ArrayList<String> l) {} m1(ArrayList l);  
    Public void m1(ArrayList<Integer> l){} m1(ArrayList l1);  
}// CE: name clash: both methods have same erasure
```

At compile time:

1. Compile code normally by considering generic syntax
2. Remove generic syntax
3. Compile once again resultant code.

Garbage collections

1. Introduction
2. The ways to make an object eligible for GC
3. The methods for requesting JVM to run garbage Collector
4. Finalization

Introduction

In old languages like C++ programmer is responsible to create new object and to destroy use less objects usually programmer taking very much care while creating objects and neglecting destruction of use less object because of his negligence at certain point for creation of new object sufficient memory may not be available (because total memory filled with use less objects only) and total application will be down with memory problems hence out of memory error is very common problem in old languages like C++.

But in java programmer is responsible only of creations of object and programmer is not responsible to destroy use less objects. Sun people provided one assistant to destroy use less objects this assistant is always running in the background (DEMON thread) and destroy use less objects just because of this assistance the chance of failing java program with memory problems is very -very low this assistant is nothing but garbage collector.

Hence the main objective of **garbage collector** is to destroy use less objects.

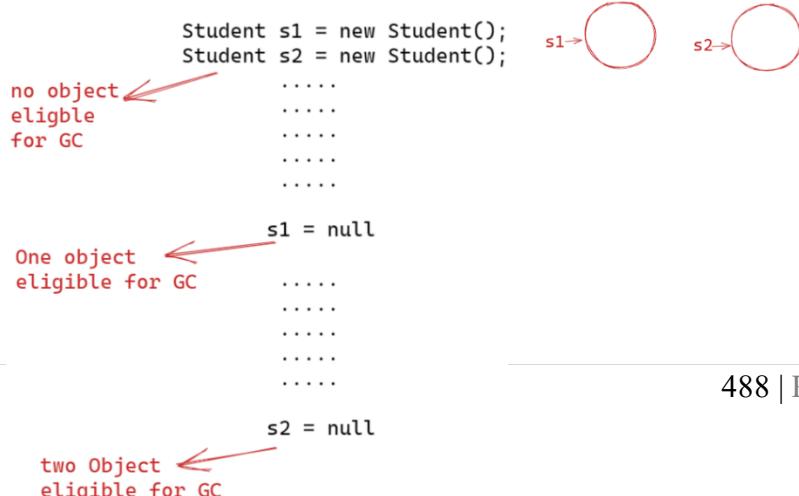
The ways to make Object eligible for GC

- Even though programmer is not responsible to destroy uses less objects it is highly recommended to make an object eligible of 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 4 ways

1. Nullifying the reference variable:

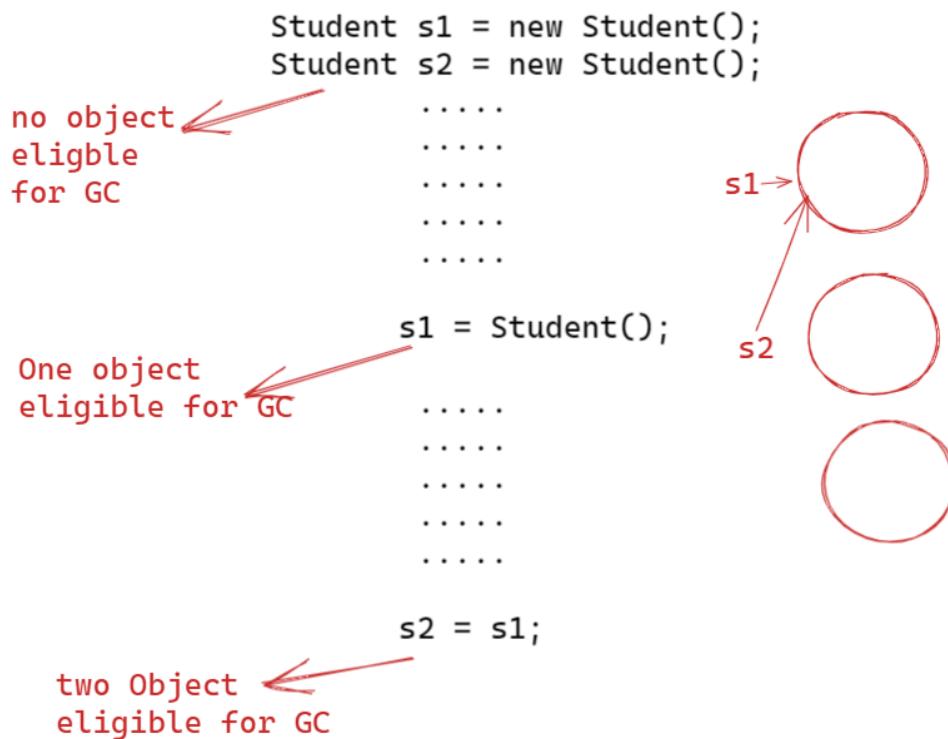
If an object no longer required then assign null to all its references variables, then that object automatically eligible of garbage collection.

This approach is nothing but nullifying the reference variable



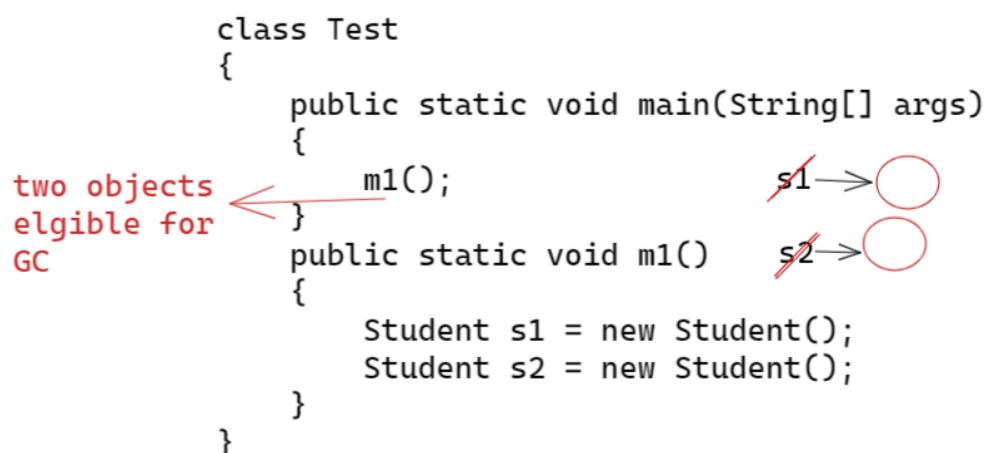
2. Reassigning the reference variable:

If an object no longer required then reassign its reference variable to some other object, then old object by default eligible for Garbage collection.

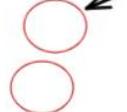


3. Objects created inside a method:

The objects which are created inside a method are by default eligible for GC once method completes



```

class Test
{
    public static void main(String[] args)
    {
        Student s = m1();
        s1 
    }
    public static Student m1() s2
    {
        Student s1 = new Student();
        Student s2 = new Student();
        return s1;
    }
}

```

Example:

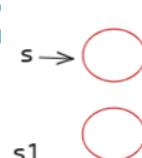
```

class Test
{
    public static void main(String[] args)
    {
        m1(); 
    }
    public static Student m1() s2
    {
        Student s1 = new Student();
        Student s2 = new Student();
        return s1;
    }
}

```

Example:

```

class Test
{
    static Student s;
    public static void main(String[] args)
    {
        m1(); 
    }
    public static void m1()
    {
        s = new Student();
        Student s1 = new Student();
    }
}

```

4. Island of Isolation:

```

class Test
{
    Test i;
    public static void main(String[] args)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        Test t3 = new test();

        t1.i = t2;
        t2.i = t3;
        t3.i = t1;

        t1 = null;
        t2 = null;
        t3 = null;
    }
}

```

Note:

- If an object doesn't contain any reference variable then it is eligible for garbage collection always.
- Even though an object has references, sometimes it is eligible for garbage collection (if all references are internal references).

Example:

Island of isolation

The ways for requesting JVM to run GC

- Once we made an object eligible for GC, it may not be destroyed immediately by the garbage collector. When ever JVM runs GC, then only the objects will be destroyed. But when exactly JVM runs garbage collector, we can't expect it is varied form JVM to JVM.
- Instead of waiting until JVM runs garbage collector, we can request a JVM to run garbage collector programmatically, but whether JVM accepts our request or not, there is no guarantee, but most of the times JVM accepts our request.
- The following are two 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 and it is a singleton class. We can create a Runtime object by using Runtime.getRuntime().

Runtime r = Runtime.getRuntime();

Once we got Runtime object we can call the following methods on that object

1. totalMemory ():

It returns number of bytes of total memory present in the heap that is heap size.

2. freeMemory ():

It returns number of bytes of free memory present in the heap.

3. gc ()

For requesting JVM to run garbage Collector.

Example:

```
import java.util.Date;
class RuntimeDemo
{
    public static void main(String[] args)
    {
        Runtime r = Runtime.getRuntime();
        System.out.println(r.totalMemory());//126877696
        System.out.println(r.freeMemory());//125535480
        for (int i = 1; i <= 10000; i++)
        {
            Date d = new Date();
            d = null;
        }
        System.out.println(r.freeMemory());//125521848
        r.gc();
        System.out.println(r.freeMemory());//126581744
    }
}
```

Note:

gc () method present in System class is a static method where as gc() method present in Runtime class is instance method

Which of the following is valid way for requesting JVM to run garbage collector

1. System.gc () => valid
2. Runtime.gc (); => invalid
3. (new Runtime()).gc() => invalid
4. Runtime.getRuntime().gc(); => valid

Note:

It is covenant to use System.gc () method when compared with Runtime class gc()

With respect to performance it is highly recommended to use Runtime class gc() method when compared with System class gc() method because System.gc() method internally calls Runtime class gc() method

Class System

```
{  
    Public static void gc ()  
    {  
        Runtime.getRuntime().gc ();  
    }  
}
```

Finalization

Just before destroying an object garbage collector calls finalize() method to perform clean-up activities.

Once finalize () methods 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 clean up activities.

Case I

Just before destroying 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 String object is eligible of gc then String class finalize () method will be executed but not Test class finalize () method

```
class FinalizeDemoCase1  
{  
    public static void main(String[] args)  
    {  
        //String s = new String("malachi");  
        FinalizeDemoCase1 s = new FinalizeDemoCase1 ();  
        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 as 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

O/P:
End of main
Finalize method called
Or
Finalize method called
End of main

Case II

Based our requirement we can call finalize () method explicitly then it will be executed just like a normal method called and object won't be destroyed.;

```

class FinalizeDemoCase2
{
    public static void main(String[] args)
    {
        FinalizeDemoCase2 t = new FinalizeDemoCase2();
        t.finalize();
        t.finalize();
        t = null;
        System.gc();
        System.out.println("end of main");
    }
    public void finalize()
    {
        System.out.println("finalize method called");
    }
}

```

In the above program finalize() method got executed three times in that two times explicitly by the programmer and one time by the garbage collector in this case output is:

O/P
finalize method called
finalize method called
end of main
finalize method called

note:

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

init (), service (), destroy () method are considered as life cycle methods of servlet. Just before destroy servlet object web container calls destroy () methods to perform clean up activities. But base on our requirements we can call destroy () method from init (), service () method then destroy () method will be executed just like a normal method call and servlet object won't be destroyed.

Case III

Even though object eligible for gc for multiple times but garbage collector calls finalize() method only once.

```
class FinalizeDemoCase3
{
    static FinalizeDemoCase3 s;
    public static void main(String[] args) throws InterruptedException
    {
        FinalizeDemoCase3 f = new FinalizeDemoCase3();
        System.out.println(f.hashCode());
        f = null;
        System.gc();
        System.out.println("end of main");
        Thread.sleep(5000);
        System.out.println(s.hashCode());
        s = null;
        System.gc();
        Thread.sleep(10000);
        System.out.println("end of main");
    }
    public void finalize()
    {
        System.out.println("finalize method called");
        s = this;
    }
}
```

Output:

```
366712642
end of main
finalize method called
366712642
end of main
```

note:

In the above program even though object eligible for gc two times but garbage collector calls finalize () method only once

Case IV

We cannot expect exact behaviour of garbage collector it is varied from JVM to JVM hence for the following questions we can't provide exact answers.

1. when exactly JVM runs garbage collector
2. in which order garbage collector identifies eligible objects
3. in which order garbage collector destroy eligible objects.
4. whether garbage collector destroy all eligible objects are not
5. what is algorithm followed by garbage collector etc.,

Note:

1. whenever programmes runs with low memory then JVM runs garbage collector but we can't except at what time
2. most of the garbage collector followed stander algorithm mark and sweep algorithm it doesn't meant every garbage collector follow the same algorithm.

```
class FinalizeDemoCase4
{
    static int count = 0;
    public static void main(String[] args) throws InterruptedException
    {
        for (int i = 0;i<1000000 ; i++)
        {
            FinalizeDemoCase4 t = new FinalizeDemoCase4();
            t = null;
        }
    }
    public void finalize()
    {
        System.out.println("finalize method called: "+ (++count));
    }
}
/*
 * If we keep on increasing this number at certain point memory problem will be raised
 * then JVM Runs GC. GC calls finalize () method on every objet separately and destroys that
 * object
 */
```

Case V

Memory leaks

The object which are not using in our program which are not eligible for GC such type of useless objects is called memory leaks

In our program if memory leaks present then the program will be terminated by raising “OutOfMemory”

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

1. HP OVO
2. Hp J Meter
3. JProbe
4. Patrol
5. IBM Tivoli

Internationalization(I18N)

The process of designing web applications in such a way that which provide support for various countries and various languages and various currency automatically without performing any change application, is called internationalization (I18N).

For example, if the requesting is coming from India then the response should be India people understandable form and if the requesting is coming from US then the response should be in US people understandable form.

We can implement internationalization by using the following three class

1. Locale
2. NumberFormat
3. DateFormat

1. Locale:

A locale object represents a geographic location (country, language or both). For example we can create a locale object to represent India. We can create a locale object to represent English language.

Locale class present in java.util package. It is a final class and it is direct child class of Object. It implements serializable and cloneable interfaces.

Constructors:

1. Locale l = new Locale (String language);
2. Locale l = new Locale (String language, String country);

Example:

```
Locale l = new Locale ("pa", "in");
          ↓           ↓
      Pa: Panjabi    in: India
```

Locale class already define some constants to represent some standard locale we can use this constant directly.

```
Locale.US
Locale.UK
Locale.ITALY
Locale.ENGLISH
```

Important methods of locale class

1. Public static Locale.getDefault();
2. Public static void Locale.setDefault(Locale l);
3. Public String getCountry (); US
4. Public String getLanguage (); en
5. Public String getDisplayCountry (); united State
6. Public String getDisplayLanguage (); English
7. Public static String [] getISOLanguages () ;
8. Public static String [] getISOCountries () ;
9. Public static Locale [] getAvailableLocales () ;

Example:

```
import java.util.*;
class LocaleDemo
{
    public static void main(String[] args)
    {
        Locale l1 = Locale.getDefault();
        System.out.println(l1.getCountry()+"----"+l1.getLanguage()); //IN----en
        System.out.println(l1.getDisplayCountry()+"----"
        "+l1.getDisplayLanguage());//India-----English
        Locale l2 = new Locale("tel","in");
        Locale.setDefault(l2);
        System.out.println(Locale.getDefault().getDisplayLanguage());//Telugu
        String[] s3 = Locale.getISOLanguages();
        for (String s4 : s3 )
        {
            System.out.println(s4);//display two character languages
        }
        String[] s4 = Locale.getISOCountries();
```

```

for (String s5 :s4 )
{
    System.out.println(s5);//display two character countries
}
Locale[] s = Locale.getAvailableLocales();
for (Locale s1 :s )
{
    System.out.println(s1.getDisplayCountry()+"-----"
"+s1.getDisplayLanguage());//display all countries and languages
}
}
}

```

2. NumberFormat

Various locations follow various styles to represent a java number.
Example:

```

Double d = 123456.789
In : 1,23,456.789
Us: 123,456.789
Italy: 123.456,789

```

We can use NumberFormat class to format a java number according to a particular Locale.

NumberFormat class present in `java.text` package and it is an abstract class.

`NumberFormat nf = new NumberFormat(); => invalid`

Getting NumberFormat object for default locale:

NumberFormat class define the following methods for these purpose

```

Public static NumberFormat getInstance ();
Public static NumberFormat getCurrencyInstance ();
Public static NumberFormat getPercentInstance ();
Public static NumberFormat getNumberInstance ();

```

Getting NumberFormat object for specific locale:

The above methods are same but we have to pass the corresponding Locale object as argument

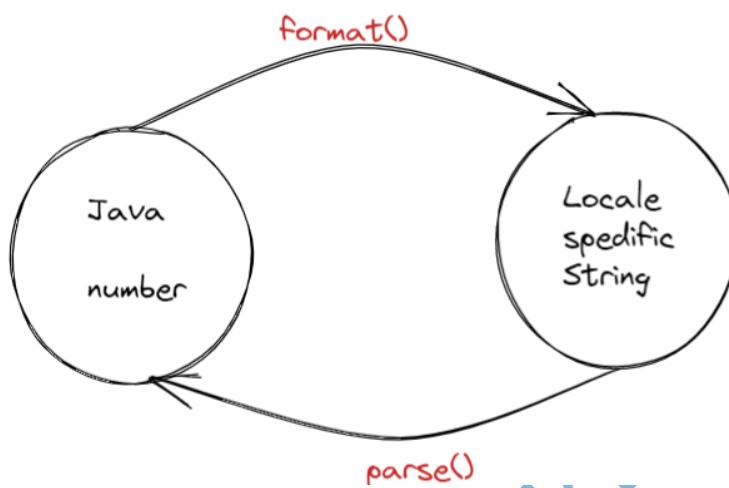
```
Public static NumberFormat getNumberInstance (Locale l);
```

Once we got NumberFormat object we can call `format ()` and `parse ()` methods on that object.

```
Public String format (long l);  
Public String format (double d);
```

To convert java number form to locale specific String form.

```
Public Number parse (String s) throws ParseException  
To convert locale specific String form to java number form.
```



Write a program to display a java number in Italy specific form

```
import java.text.*;  
import java.util.*;  
class NumberFormatDemo  
{  
    public static void main(String[] args)  
    {  
        double d = 123456.789;  
        NumberFormat nf = NumberFormat.getInstance(Locale.ITALY);  
        System.out.println("Italy Form is :" + nf.format(d)); //Italy Form is :123.456,789  
    }  
}
```

Write a program to display a java number in Italy, UK, US and India currency form

```
import java.text.*;  
import java.util.*;  
class CurrencyFormatDemo  
{  
    public static void main(String[] args)  
    {  
        double d = 123456.789;  
        Locale india = new Locale("tel", "IN");  
        NumberFormat nf = NumberFormat.getCurrencyInstance(india);  
        System.out.println("India Form is :" + nf.format(d));  
        //India Form is : INR 123,456.79
```

```

        NumberFormat nf1 = NumberFormat.getCurrencyInstance(Locale.US);
        System.out.println("US Form is :" + nf1.format(d)); //US Form is :$123,456.79

        NumberFormat nf2 = NumberFormat.getCurrencyInstance(Locale.UK);
        System.out.println("UK Form is :" + nf2.format(d)); //UK Form is :£123,456.79

        NumberFormat nf3 = NumberFormat.getCurrencyInstance(Locale.ITALY);
        System.out.println("Italy Form is :" + nf3.format(d));
                                //Italy Form is :? 123.456,79
    }
}

```

Setting Maximum and Minimum Fraction and Integer digits:

NumberFormat class defines the following methods for this purpose.

```

Public void setMaximumFractionDigits (int n);
Public void setMinimumFractionDigits (int n);
Public void setMaximumIntegerDigits (int n);
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(124.4)); //123.40

```

Case 3:

```

nf.setMaximumIntegerDigits(3);
System.out.println(nf.format(123456.789)); //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 locations follow various styles to represent date.

Example:

In: dd-mm-yyyy

US: mm-dd-yyyy

We can use DateFormat to format java Date according to particular Locale.

DateFormat class present in java.text package and it is an abstract class.

`DateForma df = new DateFormat(); => invalid`

Getting DateFormat Object for default Locale

```
Public static DateFormat getInstance ();
Public static DateFormat getDateInstance ();
Public static DateFormat getDateInstance (int style);
```

The allowed styles are 0 to 3

```
DateFormat.FULL    => 0 (Saturday 26th October 2022)
DateFormat.LONG    => 1 (26th October 2022)
DateFormat.MEDIUM =>2 (26th Oct 2022)
DateFormat.SHORT   => 3 (26/11/2022)
```

The default style is medium

Getting DateFormat object for specific Locale

```
Public static DateFormat getDateInstance (int style, Locale l);
```

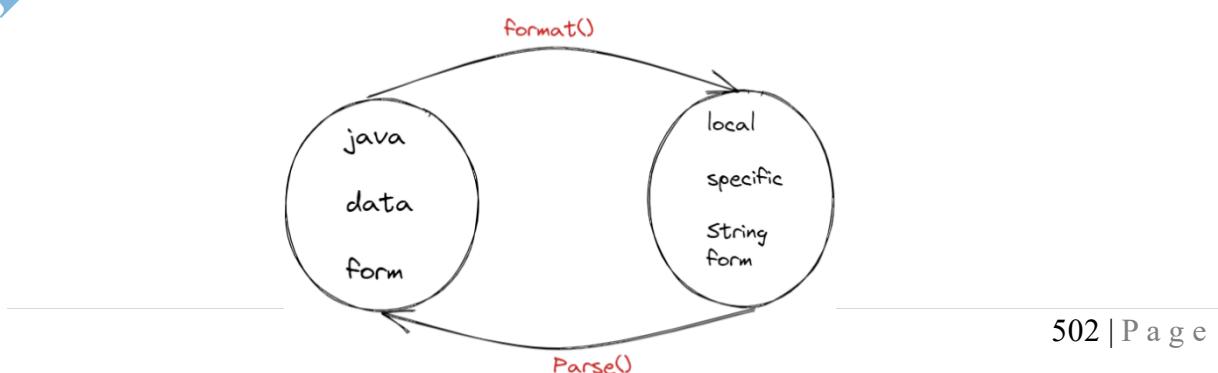
Once we got DateFormat object we can call the following methods on that object.

`Public String format (Date d)`

To convert java Date form to locale specific String form

`Public Date parse(String s) throws ParseException`

To convert Locale specific String form to java Date form



Write a program to display current system date in all possible Styles of US form

```
import java.text.*;
import java.util.*;
class DateFormatDemo
{
    public static void main(String[] args)
    {
        System.out.println("full from: "+DateFormat.getDateInstance(0,Locale.US).format(new Date())); //full from:Saturday, November 26, 2022
        System.out.println("Long from:" +DateFormat.getDateInstance(1,Locale.US).format(new Date ()));
        //Long from:November 26, 2022
        System.out.println("Medium from: "+DateFormat.getDateInstance(2,Locale.US).format(new Date()));
        //Medium from:Nov 26, 2022
        System.out.println("Short from:"+DateFormat.getDateInstance(3,Locale.US).format(new Date()));
        //Short from:11/26/22
    }
}
```

Write a program to display current system date in UK, US and ITALY styles.

```
import java.text.*;
import java.util.*;
class DateFormatDemo1
{
    public static void main(String[] args)
    {
        DateFormat uk = DateFormat.getDateInstance(0,Locale.UK);
        DateFormat us = DateFormat.getDateInstance(0,Locale.US);
        DateFormat italy = DateFormat.getDateInstance(0,Locale.ITALY);
        System.out.println("UK Style is:- "+uk.format(new Date()));
        //UK Style is:- Saturday, 26 November 2022
        System.out.println("US Style is:- "+us.format(new Date()));
        //US Style is:- Saturday, November 26, 2022
        System.out.println("ITALY Style is:- "+italy.format(new Date()));
        //ITALY Style is:- sabato 26 novembre 2022
    }
}
```

Getting DateFormat object to display both date and time

DateFormat class defines the following methods for this purpose.

```
Public static DateFormat getDateInstance ();
Public static DateFormat getDateInstance (int datestyle, int timestyle);
Public static DateFormat getDateInstance e(int datestyle, int timestyle, Locale l);
```

The allowed styles for time also: 0 to 3 only

```
import java.text.*;
import java.util.*;
class DateFormatDemo
{
    public static void main(String[] args)
    {
        DateFormat uk = DateFormat.getDateInstance(0,0,Locale.UK);
        DateFormat us = DateFormat.getDateInstance(0,0,Locale.US);
        DateFormat italy = DateFormat.getDateInstance(0,0,Locale.ITALY);
        System.out.println("UK Style is:- "+uk.format(new Date()));
            //UK Style is:- Saturday, 26 November 2022 19:24:38 o'clock IST
        System.out.println("US Style is:- "+us.format(new Date()));
            //US Style is:- Saturday, November 26, 2022 7:24:38 PM IST
        System.out.println("ITALY Style is:- "+italy.format(new Date()));
            //ITALY Style is:- sabato 26 novembre 2022 19.24.38 IST
    }
}
```

ENUM(Enumeration)

If we want to represent a group of named constants then we should go for ENUM.

Example:

```
Enum Month
{
    JAN, FEB, MAR, ...., DEC; (semicolon is optional)
}

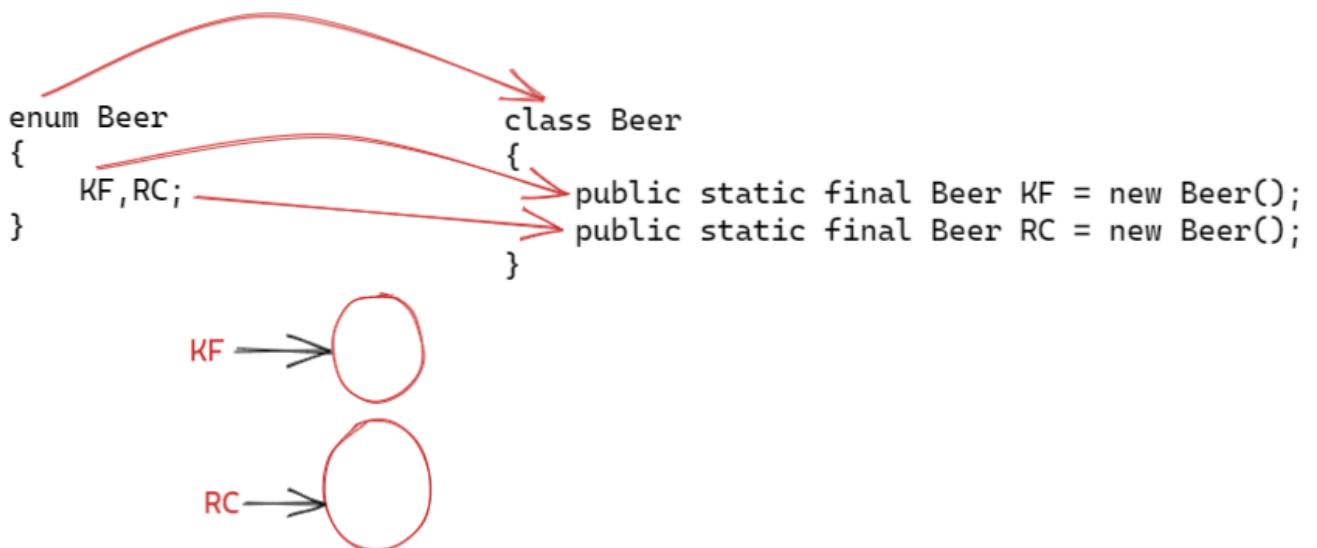
Enum Beer
{
    KF, KO, RC, FO; (semicolon is optional)
}
```

The main objective of enum is to define our own data types (enumerated data types).

Enum concept introduced in 1.5 version. When compared with old languages enum java enum is more powerful.

Internal implementation of enum:

1. Every enum is internally implemented by using class concept
2. Every enum constant is always public static final
3. Every enum constant represents an object of the type enum



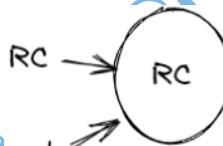
Enum declaration and usage:

Every enum constant is always **public static final** and hence we can access enum constant by using enum name

```

enum Beer
{
    KF, KO, RC, FO;
}
class EnumDemo
{
    public static void main(String[] args)
    {
        Beer b = Beer.RC;
        System.out.println(b); // RC
    }
}

```

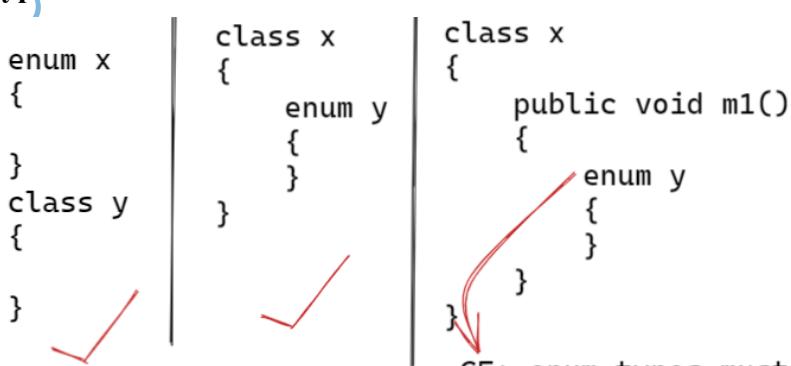


Note:

Inside enum `toString()` method internally implemented to return name of the constant.

We can declare enum either with in the class or outside of the class but not inside a method.

If we are trying to declare inside a method then we will get compile time error saying "**enum types must not be local**".



CE: enum types must not be local 505 | Page

If we declare enum outside of the class the applicable modifiers are
public,
<default>,
strictfp

If we declare enum inside a class the applicable modifiers are
public,
<default>,
strictfp
+
private,
protected
static

enum V/S switch ()

until 1.4 version the allowed argument types for the switch() statement are byte, short, char, int but from 1.5 version onwards corresponding wrapper classes and enum types are allowed. From 1.7 version onwards String type also allowed

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

Hence from 1.5 version on wards we can pass enum type as argument to switch statement.

```
enum Beer
{
    KF, KO, RC, FO;
}
class enumvsswitchDemo
{
    public static void main (String [] args)
    {
        Beer b = Beer.KF;
        switch(b)
        {
            case KF:
                System.out.println("it is childrens brand");
                break;
        }
    }
}
```

```

        case KO:
            System.out.println(" it is too light");
            break;
        case RC:
            System.out.println(" it s not that much kick");
            break;
        case FO:
            System.out.println("By one get one free");
            break;
        default:
            System.out.println("other brands are not recommended");
    }
}
}

```

If we pass enum type as argument to switch statement then every case label should be valid enum constant other wise we will get compile time error

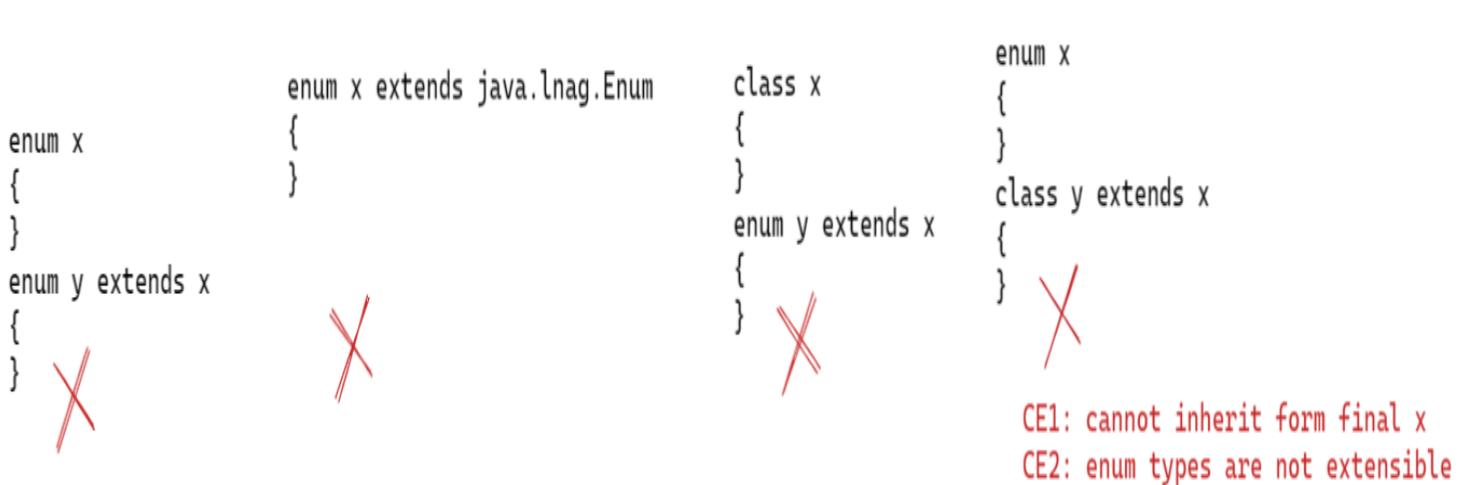
```

Switch(b)
{
    Case KF: => valid
    Case KO: => valid
    Case RC: => valid
    Case FO: => valid
    Case KALYANI: => invalid
                CE: unqualified enumeration constant name required
}

```

Enum v/s Inheritance

1. Every enum is all ways direct child class of `java.lang.Enum` and hence our enum cannot extend any other enum (because java won't provide support for multiple inheritance).
2. Every enum is all ways final implicitly and hence for our enum we cannot create child enum
Because of above reason we can conclude inheritance concept is not applicable for enum explicitly and we can't use `extend` keyword for enum.



Any way an enum can implement any number interfaces.

Interface x

```
{  
}
```

Enum y implements x

```
{  
}
```

java.lang.Enum

every enum in java is the direct child class of java.lang.Enum and hence this class acts as base class for all java Enum's.

it is an abstract class and it is the child class of object. It implements serializable and comparable interfaces.

Values () method

Every enum implicitly contains values() method to list out all values present inside enum.

```
Beer[] b = Beer.values();
```

Note:

Values() method not present in java.lang.Enum and object classes. Enum key word implicitly provides this method.

ordinal () method

inside enum order of constant is important and we can represent 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 based like array index.

```
enum Beer  
{  
    KF,KO,RC,FO;  
}  
class ValuesMethoDemo  
{  
    public static void main(String[] args)  
    {  
        Beer[] b = Beer.values();  
        for (Beer b1: b )  
        {  
    }
```

```

        System.out.println(b1+" ---- "+b1.ordinal());
    }
}
/*
OUTPUT:
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 constants we can take methods, constructor, normal variables etc., hence java enum is more power full then old languages enum

Even in side java enum we can declare main method and we can run enum class directly form command prompt.

```

enum EnumFishDemo
{
    STAR, GUPPY, GOLD; (semicolon is mandatory)
    public static void main(String[] args)
    {
        System.out.println("Enum main method");
    }
}
Javac EnumFishDemo
Java EnumFishDemo
Output:
Enum main method

```

Note:

In addition to constant if we are taking extra member like a method then list of constant should be in the first line and should ends with semicolon(;)

Example:

```

enum EnumFishDemo
{
    STAR, GUPPY, GOLD; ( semicolon Is mandatory)
    public static void main(String[] args)
    {
        System.out.println("Enum main method");
    }
} => valid

```

```

enum EnumFishDemo
{
    STAR, GUPPY, GOLD
    public static void main(String[]
args)
    {
        System.out.println("Enum
main method");
    }
} => invalid because semicolon is not
there

```

```

enum EnumFishDemo
{
    public static void main(String[]
args)
    {
        System.out.println("Enum
main method");
    }
    STAR, GUPPY, GOLD;
} => invalid

```

Inside enum if we are taking any extra member like a method compulsory first line contains list of constant at least semicolon

```

enum EnumFishDemo
{
    public void m1()
    {

    }
} => invalid

```

```

enum EnumFishDemo
{
    ;
    public void m1()
    {

    }
} => valid

```

Any way an empty enum is valid java syntax.

```

enum fish
{
}

} => valid

```

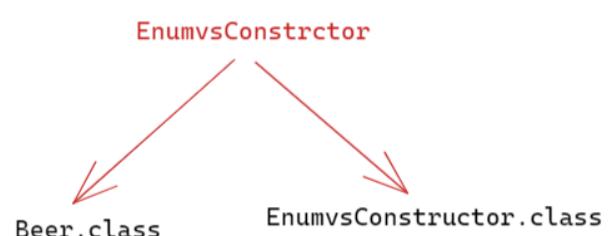
Enum v/s constructor

An enum can constructor. Enum constructor will be executed separately for enum constant at the time of enum class loading automatically.

```

enum Beer
{
    KF,KO,RC,FO;
    Beer()
    {
        System.out.println("enum constructor");
    }
}
class EnumvsConstrctor
{
    public static void main(String[] args)
    {
        Beer b = Beer.RC; =====> 1
        System.out.println("Hello main method");
    }
}

```



```

        }
    }
/*
output:
enum constructor
enum constructor
enum constructor
enum constructor
Hello main method
*/

```

If we comment line one then the output is “Hello main method”;

We can't create enum object directly and hence we can't invoke enum constructor directly.

Beer b = new Beer() => invalid
CE: enum type may not be instantiated.

Example:

```

enum Beer
{
    KF(70),KO(80),RC(90),FO;
    int price;
    Beer(int price)
    {
        this.price = price;
    }
    Beer()
    {
        this.price = 65;
    }
    public int getPrice()
    {
        return price;
    }
}
class FullFunctionEnum
{
    public static void main(String[] args)
    {
        Beer[] b = Beer.values();
        for(Beer b1:b)
        {
            System.out.println(b1+"----"+b1.getPrice());
        }
    }
}

```

```

/*
output
KF----70
KO----80
RC----90
FO----65
*/

```

Note:

KF => public static final Beer KF = new Beer();
KF (70) => public static final Beer KF = new Beer(70);

Note:

Inside enum we can declare methods but should be concrete methods only and we can't declare abstracts methods.

Case 1:

Every enum constant represents an object of the type enum hence whatever methods we can apply on normal objects, can be applicable or enum constants also

Example:

Beer.KF.equals(Beer.RC) => valid
Beer.KF.hashCode() > Beer.RC.hashCode(); => valid
Beer.KF < Beer.RC => invalid
Beer.KF.ordinal() < Beer.RC.ordinal(); => valid

Case 2:

If we want to use any class or interface directly from outside package then the required import is normal import.

If we want to access static members with out class name then the required import is static import.

```
import static java.lang.Math.sqrt;
import java.util.ArrayList;
class Test
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        System.out.println(sqrt(4));
    }
}
```

```
package pack1;
public enum Fish
{
    STAR,GUPPY;
}
```

```
package pack2;
public class Test1
{
    public static void main(String[] args)
    {
        Fish f = Fish.GUPPY;
        System.out.println(f);
    }
}

the required import is:
import pack1.Fish;
(or)
import pack1.*;
```

```
package pack3;
public class Test2
{
    public static void main(String[] args)
    {
        System.out.println(STAR);
    }
}

the requiered import is:
import static pack1.Fish.STAR;
(or)
import static pack1.Fish.*;
```

```
package pack4;
public class Test3
{
    public static void main(String[] args)
    {
        Fish f = Fish.STAR;
        System.out.println(GUPPY);
    }
}

the required imports are:
import pack1.Fish;
(or)
import pack1.*;

import static pack1.Fish.GUPPY;
(or)
import static pack1.Fish.*;
```

Case 3

Example1:

```
enum Color
{
    BLUE,RED,GREEN;
    public void info()
    {
        System.out.println("universal color");
    }
}
class EnumCase2Demo
{
    public static void main(String[] args)
    {
        Color[] c = Color.values();
```

```

        for(Color c1 : c)
        {
            c1.info();
        }
    }

/*
output:
universal color
universal color
universal color
*/

```

Example 2:

```

enum Color
{
    BLUE, RED
    {
        public void info()
        {
            System.out.println("Dangerous color");
        }
    }, GREEN;
    public void info()
    {
        System.out.println("universal color");
    }
}
class EnumCase2Demo1
{
    public static void main(String[] args)
    {
        Color[] c = Color.values();
        for(Color c1 : c)
        {
            c1.info();
        }
    }
/*
output:
universal color
Dangerous color
universal color
*/

```

What is the difference between enum vs Enum vs Enumeration

enum:

enum is a keyword in java which can be used to define a group of name constants.

Enum:

Enum is a class in java present in java.lang package. Every Enum in java should be direct child class of Enum class hence this class act as base class for all java Enum's

Enumeration:

Enumeration is an interface present in java.util package. We can use Enumeration object to get objects one by one from the collection.

Development

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

[options]:

- version
- d
- source
- verbose
- cp/-classpath
- ...
-

Java:

We can use java command to run a single class file.

Java [options] Test A B C

ABC: are command line arguments

[options]:

- version
- D
- cp/ -classpath
- ea/-era/-dsa/-da

Note:

We can compile any number of source file at a time but we can run only one dot(.) .class file at a time.

ClassPath:

ClassPath describe the location where required .class files are available . java compiler and JVM will use ClassPath to locate required .class file

By default, JVM will always searches in current working directory for required .class file. If we set class path explicitly then JVM will search in our specified ClassPath location and JVM won't search in current working directory.

We can set the class path in 3 ways

1. by using environment variable ClassPath

This way of setting of ClassPath is permanent and will be preserved across system restarts.

When we are installing a permanent software in our system then this approach is recommended.

2. At command prompt level by using set command

Set ClassPath = d:\java programmes

This way of setting class path will be preserved only for particular command prompt once command prompt closes automatically class path will be lost.

3. At command level by using -cp option

Java -cp d:\java programmes Test

This way of setting ClassPath will be preserved only for particular command once command execution completes automatically class path will be lost.

Note:

Among three ways of setting ClassPath, setting ClassPath at command level is recommended because dependent classes 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 in the specified ClassPath location only.

Example 1:

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

D:\java_programes> javac Test.java

D:\java_programes> java Test
o/p: Hello world

D:\>java Test
RE: NoClassDefFoundError: Test

D:\> java -cp d:\java_programes Test
o/p: Hello world

c:\. Java -cp d:\java_programes Test
o/p: Hello world

E:\>java -cp d:\java_programes Test
o/p: Hello world

d:\java_programes>java -cp E: Test
RE: NoClassDefFoundError: test

d:\java_programes> java -cp ..; E: Test
o/p: Hello world

Example 2:

```
C:      => AStudent .class
public class AStudent
{
    public void m1()
    {
        System.out.println("I want JOB Immediately");
    }
}
```

```
D:          => ItIndustry.class
public class ItIndustry
{
    public static void main(String[] args)
    {
        AStudent a1 = new AStudent();
        a1 = m1();
        System.out.println("You will get soon!!");
    }
}
```

C:> java AStudent.java

```
D:> javac ItIndustry.java
CE: cannot find symbol
Symbol: class AStudent
Location: class ItIndustry
```

D:> javac -cp c: ItIndustry.java

```
D:> java ItIndustry
RE: NoClassDefFoundError: AStudent
```

```
D:>java -cp c: ItIndustry
RE: NoClassDefFoundError: ItIndustry
```

D:> java -cp .;C: ItIndustry

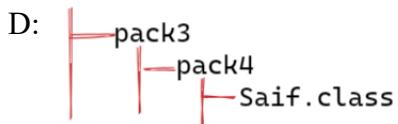
E:> java -cp D;;C: ItIndustry

Example 3:

C:

```
    ┌── pack1
    │   ┌── pack2
    │   └── Kareena.class
```

package pack1.pack2;
public class Kareena
{
 public void m1()
 {
 System.out.println("Hello Saif , can you plz set HelloTune");
 }
}



```

Package pack3.pack4;
Import pack1.pack2.Kareena;
Public class Saif
{
    Public void m2()
    {
        Kareena k = new Kareena();
        k.m1();
        System.out.println("not possible because I am in SCJP class");
    }
}
  
```

E: => Bhupathi.class

```

Import pack3.pack.Saif;
Public class Bhupathi
{
    Public static void main(String[] arg)
    {
        Saif s = new Saif();
        s.m2();
        System.out.println("Hello Kareena can I Help you");
    }
}
  
```

C:\> javac -d . Kareena.java

D:\> javac -d . Saif.java
CE: cannot find symbol
Symbol: class Kareena
Location: class pack3.pack4.Saif

D:\> javac -d . -cp c: Saif.java

E:\> javac Bhupathi.java
CE: cannot find symbol
Symbol: class Saif
Location: class Bhupathi

E:\> javac -cp d: Bhupathi.java

E:\> java Bhupathi
RE: NoClassDefFoundError; pack3.pack4.Saif

E:\> java -cp .; D: Bhupathi
RE: NoClassDefFoundError:pack1.Pack2.Kareena

E:> java -cp .;D: ; C: Bhupathi

F:> java -cp E:; D:; C: Bhupathi

Conclusions:

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 one level dependency where as JVM will check all levels of dependency.
3. In class path the order of location is important JVM is always consider from left to right until required match is available.

Example:

C:

```
public class Renuka
{
    public static void main(String[] arg)
    {
        System.out.println("c: Renuka");
    }
}
```

D:

```
Public class Renuka
{
    public static void main(String[] arg)
    {
        System.out.println("D: Renuka");
    }
}
```

E:

```
Public class Renuka
{
    public static void main(String[] arg)
    {
        System.out.println("E: Renuka");
    }
}
```

```
java -cp D;; C;; E: Renuka;
o/p D: Renuka
```

```
java -cp E;; D;; C;; Renuka
o/p: E:Renuka
```

jar file:

A group of .class files is nothing but jar file or Zip file

All third-party software plugins are default available in the form of jar files only.

Example 1:

To develop a servlet all dependent class are available in servlet-api.jar we have to place this jar file in ClassPath to compile a servlet program.

Example 2:

To run a JDBC program all dependent class are available in OJDBC14.jar to run JDBC program we have to place this jar file in ClassPath

Example 3:

To use Log4j in our application dependent classes are available in Log4j.jar we have to place this jar file in the ClassPath then only Log4j based application can run.

1. To create a Jar file (zip file):

```
Jar -cvf bhupathicalc.jar Test.class  
Jar -cvf bhupathicalc.jar A.class      B.class      C.class  
Jar -cvf bhupathicalc.jar *.class  
Jar -cvf bhupathicalc.jar *.*
```

2. To extract a jar file (unzip file):

```
Jar -xvf bhupathicalc.jar
```

3. To display table of contents

```
Jar -tvf bhupathicalc.jar
```

Service provider role

```
public class BhupathiColorfulCalc  
{  
    public static void add(int x, int y)  
    {  
        System.out.println(x * Y);  
    }  
    public static void multiply(int x, int y)  
    {  
        System.out.println(2 * X * Y);  
    }  
}
```

```
Javac BhupathiColorfulCalc.java
```

```
Jar -cvf BhupathiCalc.jar BhupathiColorfulCalc.jar
```

Client's role

He downloaded jar file and he place in D: of client's machine

Class Bakara

```
{  
    Public static void main(String[] args)  
    {  
        BhupathiColorfulCalc.add(10,20);  
        BhupathiColorfulCalc.multiply(10,20);  
    }  
}
```

C:\java_programes> javac Bakara.java => not compile we will get CE

C:\java_programes> javac -cp D: Bakara.java => not compile

C:\java_programes> javac -cp D:\BhupathiCalc.jar Bakara.java

C:\java_programes> java Bakara

RE: NoClassDefFoundError: BhupathiColofulCalc

C:\java_programes> java -cp D: Bakara

RE: NoClassDefFoundError: Bakara

C:\java_programes> java -cp .;D: Bakara

RE: NoClassDefFoundError: BhupathiColorfulCalc

C:\java_programes> java -cp .;d:\bhupathicalc.jar Bakara

o/p: 200

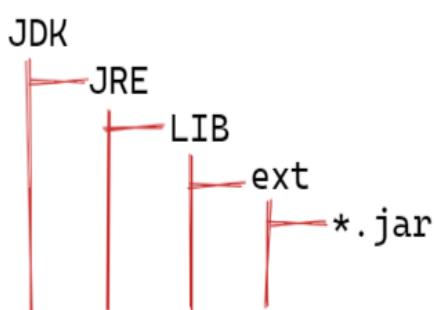
400

Note:

To place .class file in ClassPath Just location is enough but to make jar file available in ClassPath location is not enough compulsory we have to include name of the jar file also.

Short-cut way to place jar file in ClassPath

If we place jar file in the following location then all classes and interface present in the jar file by default available to java compiler and JVM we are not required to set ClassPath explicitly



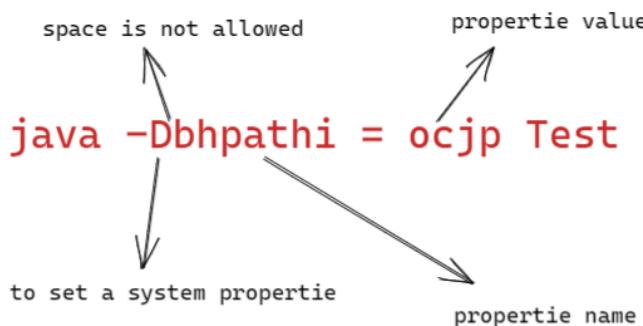
System properties

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 SystemPropertieDemo
{
    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.

```
class SystemPropertyDemo2
{
    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");
        }
    }
}
Javac SystemPropertyDemo2.java
Java -Dcourse=scjp SystemPropertyDemo2
o/p: Scjp infofomation
```

- 1. jar vs war vs ear**
- 2. web application vs enterprise application**
- 3. web server and application server**
- 4. classpath vs path**
- 5. JDK vs JRE vs JVM**
- 6. java vs javaw vs javaws**
- 7. how to create executable jar file**
- 8. in how many ways we can run a java program**

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 applications which contains servlets, jsp's, html pages, JavaScript files etc., the main advantage of maintaining web applications in the form of war file is project deployment, project delivery and project transportation is easy.

Ear (enterprise Archive):

An Ear file represents one enterprise application which contains servlets, JSP's EJB's, JMS components etc.,

Note:

In general ear file represent a group of war files and jar files.

2. Web Application v/s Enterprise Application

A web application can be developed by only web related technologies like servlet, JSP's, HTML, CSS etc.,

Example;

Online library management system
Online shopping cart

An enterprise application can be developed by any technology from java/j2EE like servlets, JSP's, EJB, JMS components et.c

Example:

Banking application
Telecom based project

Note:

J2EE/JEE compatible application is enterprise application.

3. Web server vs application server

Web server provides environment to run web application. Web server provides support for web related technologies like Servlets, JSP's, HTML etc.,

Example:

Tomcat

Application server provides environment to run enterprise applications. Application server can provide support for any technology from java/j2ee like servelets, JSP's, EJB, JMS components etc.

Example:

WebLogic
WebSphere
JBoss etc.,

Note:

Every application server contains in-built web server to provide support for web related technologies.

J2EE compatible server is application server.

4. How to create executable Jar File

JarDemo2.java

```
import java.awt.*;  
import java.awt.event.*;  
public class JarDemo2  
{  
    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);  
    }  
}
```

Manifest.MF

Main-class: JarDemo2

—

Execution form command prompt

Javac JarDemo.java
 JarDemo2.class
 JarDemo2\$1.class

Jar -cvfm JarFileforJarDemo2.jar manifest.mf JarDemo2.class JarDemo2\$1.class

Java -jar JarFileforJarDemo2.jar

Even we can run jar file by double clicking.

4. How many ways to run a java program:

We can run a java program in the following ways||

1. For command prompt we can run .class file with java command
Eg:

 Java JarDemo2

2. From command prompt we can run jar file with java command
Eg:

 Java -jar jarfileforjarDemoe.jar

3. By double clicking a batch file
4. By double clicking a jar file

4. Batch file:

A batch file contains a group of commands. When ever we a double click a batch file then all commands will be executed one by one in the sequence

java -cp d:\java_programes JarDemo2

abc.bat

5. Difference between Path and ClassPath

ClassPath:

ClassPath describe the location where required .class files are available

Java compiler and JVM will use ClassPath to locate required class files. If we are not setting ClassPath then our program may not compile and may not run.

Path:

Path describe the location where required binary executables are available. If we are not setting path then javac and java commands won't work.

Example:

Set path = C:\Program Files\Java\jdk1.8.0_202\bin

6. Difference between JDK, JRE and JVM

JDK (java development kit):

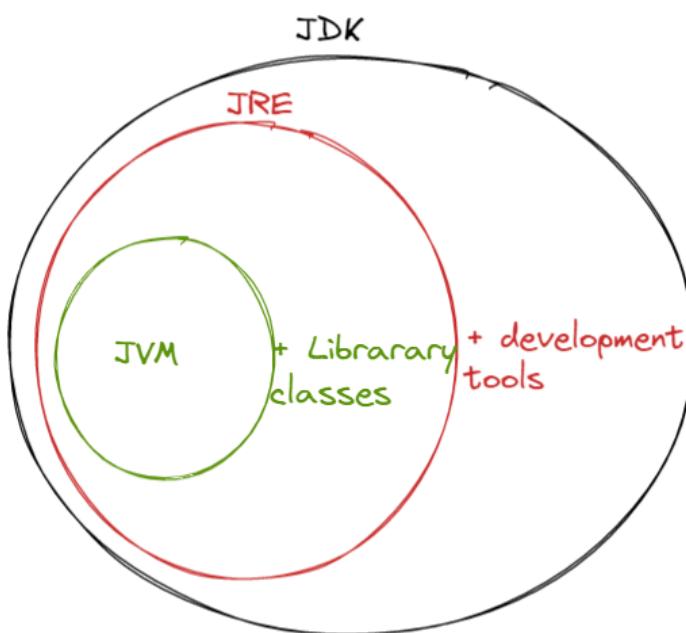
JDK provides environment to develop and run java application.

JRE (java runtime environment):

JRE provides environment to run java application.

JVM (java virtual machine):

JVM is responsible to run java program line by line hence it is an interpreter



JDK = JRE + Development Tools
JRE = JVM + Library Classes

JVM is the part of JRE where as JRE is the part of JDK

Note:

On the developers machine we have to instal JDK where as and the client machine we have to instal JRE.

Difference between java vs javaw vs javaws

Java:

We can use java command to run a java class file where SOP'S will be executed and corresponding output will be display to the console.

Javaw (java without console output):

We can use java javaw command to run a java class file where SOP's will be executed but the corresponding output won't be display 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 start its execution.
We can use javaws command as fallows

Javaws Jnlp url

- It downloads the application from the specified URL and start execution.
- The main advantage in this approach is every end user will get updated version and enhancement will become easy because of centralized control.

Assertions (1.4 version)

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 Error.

Introduction:

Very common way of debugging is usage of SOP (System.out.println) statements for the problem with SOP is after fixing the bug compulsory we have to delete SOP statement other wise these SOPs will be executed at runtime for every request, which creates performance problems and disturbs server logs.

To over come this problem sun people introduced Assertion concepts in 1.4 version.

The main advantage of Assertions when compared with SOPs is after fixing the bug, we are not required to remove assert statements they won't be executed by default at runtime based on our requirement we can enable and disable Assertions and by default Assertions are disable.

Hence the main object of Assertions is to perform debugging.

Usually, we can perform debugging in developer and testing environment but not in production environment hence assertion concept applicable for development and test environment but not for production.

Assert as keyword and identifier

assert keyword introduced in 1.4 version hence 1.4 version onward we can't use assert as identifier other wise we will get compile time error.

```
class AssertDemo
{
    public static void main(String[] args)
    {
        int assert = 10;
        System.out.println(assert);
    }
}
```

Javac AssertDemo.java

CE: as of release 1.4 'assert' is a keyword, and may not be used as an identifier (use -source 1.3 or lower to use 'assert' as an identifier)

Javac -source 1.3 AssertDemo.java
Code compiles fine but with warning

Java AssertDemo.java
o/p: 10

javac -source 1.2 AssertDemo.java => valid

javac -source 1.3 AssertDemo.java => valid

javac -source 1.4 AssertDemo.java => invalid

javac -source 1.5 AssertDemo.java => invalid

Note:

=> 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.

=> we can compile a java program according to a particular version by using -source option.

Types of assert statements

There are two 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 is satisfying and hence rest of the program will be executed normally.

If b is false then our then our assumption fails that is some where some thing goes wrong and hence the program will terminated abnormally by raising assertion error. Once we got assertion error, we will analyse the code and we can fix the problem.

```
class AssertDemo1
{
    public static void main(String[] args)
    {
        int x = 10;
        ::::::::::::::::::::;
        assert(x > 10);
        ::::::::::::::::::::;
        System.out.println(x);
    }
}
```

```

}

javac AssertDemo1.java
java AssertDemo1
    o/p: 10
java -ea AssertDemo1
    RE: AssertionException

```

Note:

By default, assert won't be executed because assertions are disabled by default but we can enable assertions by using -ea option

2. Augmented version

We can augment some description with assertion error by using augmented version.

Syntax: **assert(b): e;**

B: should be boolean

E: can be any type but most we can take String type

```

class AssertDemo2
{
    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 AssertDemo2.java
java AssertDemo2
    o/p: 10
java -ea AssertDemo2
    REAssertionError: here x value should be > 10 but it is not
*/

```

Conclusion 1:

Assert(b):e;

e will be executed if and only if first augment is false that is if the first augment is true then second argument won't be evaluated.

```

class ArgumentConclusion1
{
    public static void main(String[] args)

```

```

{
    int x =10;
    ::::::::::::::::::::
    assert(x == 10):++x;
    ::::::::::::::::::::
    System.out.println(x);
}

```

Javac AgumentConculsion1.java

Java AgumentConculsion1

o/p: 10

java -ea AgumentConculsion1

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 other wise we will get compile time error.

```

class AgumentConculsion2
{
    public static void main(String[] args)
    {
        int x =10;
        ::::::::::::::::::::
        assert(x > 10):m1();
        ::::::::::::::::::::
        System.out.println(x);
    }
    public static int m1()
    {
        return 777;
    }
}

```

Javac AgumentConculsion2.java

Java AgumentConculsion2

o/p: 10

java -ea AgumentConculsion2

o/p: AssertionException: 777

if m1() method return type is void then we will get compile time error saying “void type not allowed here”.

Note:

Among of two versions of Assertions it is recommended to use augmented version because it provides more information for debugging.

Various possible Runtime Flags

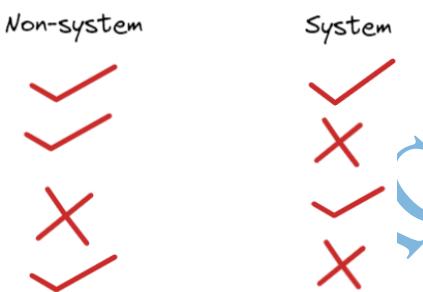
1. -ea | -enableassertions
To enable assertions in every non system class (our own classes)
2. -da | -disableassertions
To disable assertions in every non system class
3. -esa | -enablesystemassertions
To enable assertions in every system class (predefined classes)
4. -dsa | -disablesystemassertions
To disable assertions in every system class

Note:

We can use above flags simultaneously then JVM will consider these flags from left to right

Example;

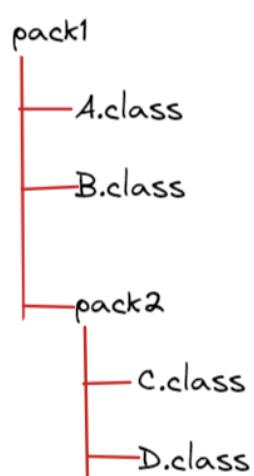
Java -ea -esa -ea -dsa -da -esa -ea -dsa test



At the end assertions will be enabled in every non-system class and disable in every system class.

Case study:

1. to enable assertions only in B class
Java -ea:pack1.B
2. to enable assertions in both B and D classes
Java -ea:pack1.B -ea:pack1.pack2.D
3. to enable assertions in 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 pack 1 except pack2 classes
Java -ea:pack1... -da:pack1.pack2...



Note:

We can enable and disable assertions either class wise or package wise also.

Cases:

1. It is always inappropriate to mix programming logic with assert statement there in guaranty for the execution of assert statement always at runtime

Example:

```
Public void withdraw(double amount)
{
    If(amount < 100)
        Throw new
IllegalRequestException();
    Else
        Process request
} => appropriate way
```

```
Public void withdrawl (double amount)
{
    Assert(amount >= 100);
    Process request;
} => inappropriate way
```

2. while performing debugging in our program if there is any place where the control is not allowed to reach that is the best place to use assertions

Example:

```
Switch(x)
{
    Case1:
        System.out.println("Jan");
        Break;
    Case2:
        System.out.println("feb");
        Break;
    Case3:
        System.out.println("decb");
        Break;
    Default:
        Assert(false);      RE: assertion error
}
```

3. it is always inappropriate for validating public method arguments by using assertions because out side person doesn't aware whether assertions are enabled or disable 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 disable in our system.

5. it is always inappropriate for validating command line arguments by using assertions because these are arguments to main method, which is public.

```

class AssertionSample
{
    int z = 5;
    public void m1(int x )
    {
        assert(x>10); => 1 //in appropriate
        switch(x)
        {
            case 10:
                System.out.println(10);
                break;
            case 11:
                System.out.println(11);
                break;
            default:
                assert(false); => 2 // appropriate
        }
    }
    private void m2(int x)
    {
        assert(x<10); => 3 //appropriate
    }
    private void m3()
    {
        assert(m1()); => 4 // inappropriate
    }
    private boolean m4()
    {
        z = 6;
        return true;
    }
}

```

Exam points of question

Class one

```

{
    public static void main(String[] args)
    {
        Int assert = 10;
        System.out.println(assert);
    }
}

```

1. javac -source 1.3 One.java
Compiler fine but with warning.

2. javac -source 1.4 One.java
CE

Class Two

```
{  
    Public static void main (String[] args)  
    {  
        Int x = 10;  
        Assert(x>10);  
    }  
}
```

3. javac -source 1.3 Two.java

CE

4. javac -source 1.4 Two.java

Compiles fine without warning.

EXAMPLE:

Class Test

```
{  
    Public static void main(String[] args)  
    {  
        Boolean assertOn = false;  
        Asseret(assertOn):asertOn = true;  
        If(assertOn)  
        {  
            System.out.println("assertOn");  
        }  
    }  
}
```

If assertions are not enable?

No output

If assertions are enabled?

RE: AssertionErroe:true

EXAMPLE:

Class Test

```
{  
    Public static void main(String[] args)  
    {  
        Boolean assertOn = true;  
        Asseret(assertOn):asertOn = false;  
        If(assertOn)  
        {  
            System.out.println("assertOn");  
        }  
    }  
}
```

If assertions are not enable?

o/p: assertOn

If assertions are enabled?

o/p: assertOn

AssertionError:

It is the child class of error and hence it is unchecked. If assert statement fails (that is argument is false) then we will get AssertionError.

Even though it is legal to catch AssertionError but it is not a good programming practice

Example:

Class Test

```
{  
    Public static void main(String[] args)  
    {  
        Int x = 10;  
  
        Try  
        {  
            Assert(x >10);      => RE: AssertionError  
        }  
        Catch (AssertionError e)  
        {  
            System.out.println(" I am stupid, because I am catching  
AssertionError");  
        }  
        System.out.println(x);  
    }  
}
```

From command prompt we have to use

Java -ea Test;

Note:

In the case of web applications if we run java program in debug mode automatically assert statements will be executed.

JVM architecture

1. Virtual machine
2. Type of virtual machines
 - a. Hardware based VM
 - b. Application based VM
3. Basic architecture of JVM
4. Class loader subsystem
 - a. Loading
 - b. Linking
 - c. Initialization
5. Types of class loaders
 - a. Bootstrap class loader
 - b. Extension class loader
 - c. Application class loader
6. How class loader works
7. What is the need of customized class loader
8. Pseudo code for customized class loader
9. Various memory areas of JVM
 - a. Method area
 - b. Heap area
 - c. Stack area
 - d. PC registers
 - e. Native method stacks
10. Program to display heap memory statistics
11. How to set maximum and minimum heap size
12. Execution engine
 - a. Interpreter
 - b. JIT compiler
13. Java native interface(JNI)
14. Complete architecture diagram of JVM
15. Class File Structure

Virtual Machine

It is a software simulation of a machine which can perform operations like a physical machine.

There are two types of virtual machines

1. Hard ware base or system based virtual machine
2. Application base or process based virtual machine

Hardware based or system based virtual machine

It provides several logical systems on the same computer with strong isolation from each other that is on one physical machine we are defining multiple logical machines.

The main advantage of hardware based virtual machine is hardware resources sharing and improves utilization hardware resources.

Example:

KVM [kernel based virtual machine for Linux system]
VMware, Xen, Cloud computing

Application based or process base virtual machine

These virtual machine acts as runtime engines to run a particular programming language applications.

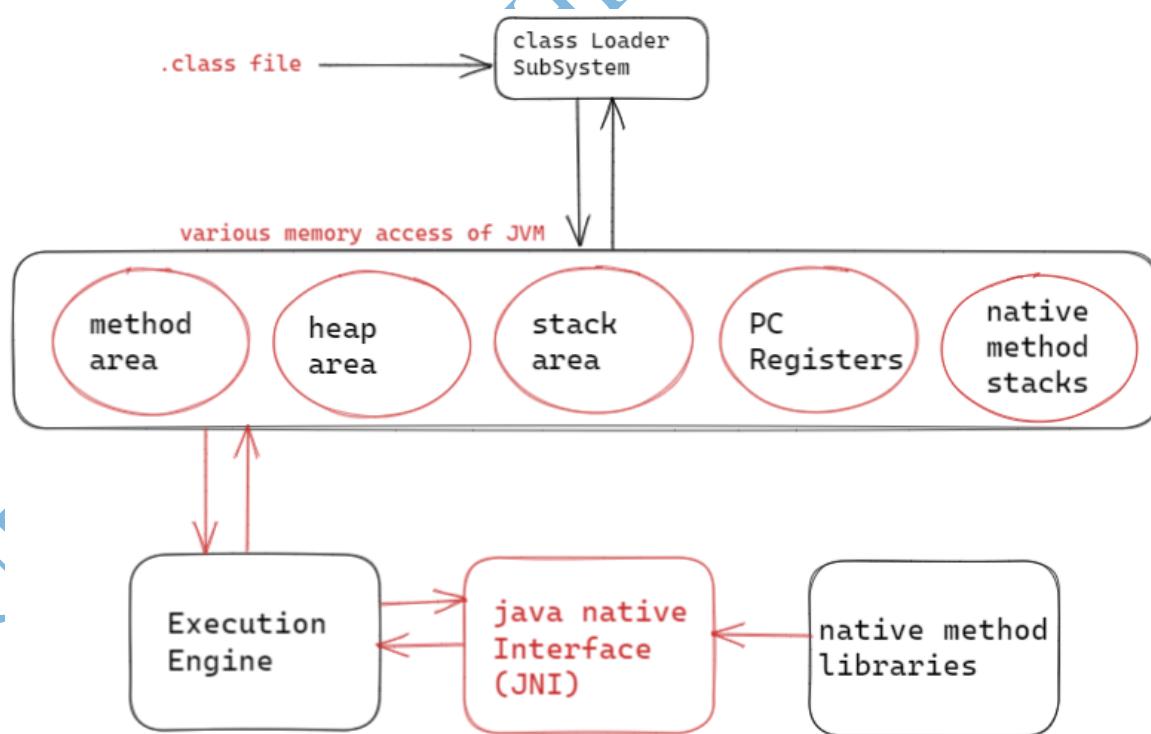
Example 1:

JVM (java virtual machine) acts as runtime engine to run java-based applications
PVM (parrot virtual machine) act as runtime engine to run Peral base applications (scripting language)
CLR (common language runtime) act as runtime engine to run dot net based applications

JVM (java virtual machine)

JVM is the part of JRE and it is responsible to load and run java class files.

Basic architecture diagram of JVM



Class loader sub-system:

Class loader subsystem is responsible for the following three activities.

1. Loading
2. Linking
3. Initialization

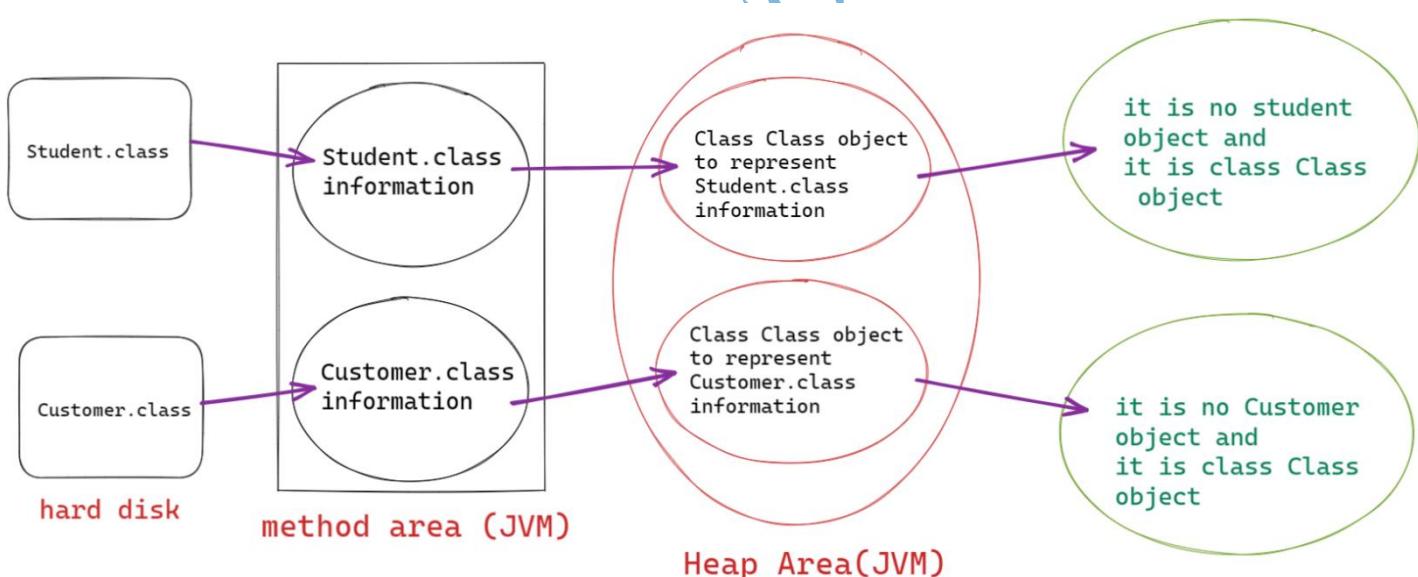
1. Loading:

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. constructors' information
6. modifiers information
7. constant pool information etc.,

After loading .class file immediately JVM creates an object for that loaded class on the heap memory of type java.lang .Class



The class, class object can be used by programmer to get class level information like methods information or variable information, constructor information etc.,

```
import java.lang.reflect.*;  
class Student  
{  
    public String getName()  
    {  
        return null;  
    }
```

```

        public int getRollNo()
        {
            return 10;
        }
    }
class ClassClassDemo
{
    public static void main(String[] args) throws Exception
    {
        int count = 0;
        Class c = Class.forName("Student");//java.lang.String , java.lang.Object
        Method[] m = c.getDeclaredMethods();
        for(Method m1:m)
        {
            count++;
            System.out.println(m1.getName());
        }
        System.out.println("number of methods: "+count);
    }
}

```

OUTPUT:

```

getName
getRollNo
number of methods: 2

```

NOTE:

For every loaded type only one class object will be created even though we are using the class multiple times in our program.

```

import java.lang.reflect.*;
class ClassClassDemo2
{
    public static 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());
        System.out.println(c2.hashCode());
        System.out.println(c1 == c2);
    }
}

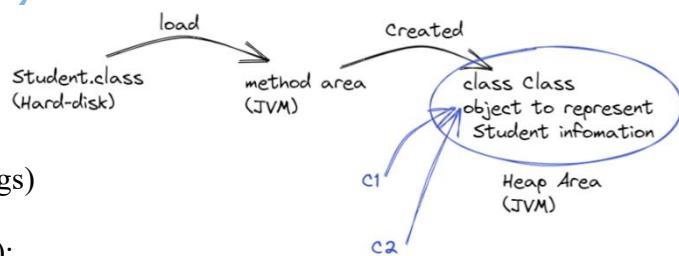
```

Ouput:

```

2018699554
2018699554
True

```



In the above program we are using student class multiple times only one class, class object got created.

2. Linking

Linking consists of 3 activities

1. verify / verification
2. prepare / preparation
3. resolve

Verification or verify:

It is the process of ensuring that binary representation of a class is structurally correct or not that is JVM will check whether the .class is generated by valid compiler or not that is whether .class file is properly formatted or not.

Internally byte code verifier is responsible for this activity. Byte code verifier is the part of class loader subsystem.

If verification fails, then we will get runtime exception saying “java.lang.VerifyError”

Preparation or prepare:

In this phase JVM will allocate memory for class level static variables and assign default values.

Note:

In initialization phase original values will be assigned to the static variables and here only default values will be assigned

Resolution or resolve:

It is the process of replacing symbolic names in our program with original memory references from method area.

Example:

```
class ResloveDemo {  
    public static void main(String[] args) {  
        String s= new String("malachi");  
        Student s1 = new Student();  
    }  
}
```

For the above class, class loader loads ResloveDemo.class, String.class, and object.class

The names of these classes are stored in constant pool of ResloveDemo class.

In resolution phase these names are replaced with original memory level reference from method area.

3. Initialization:

In this all-static variable are assigned with original values and static blocks will be executed from parent to child and from top to bottom.

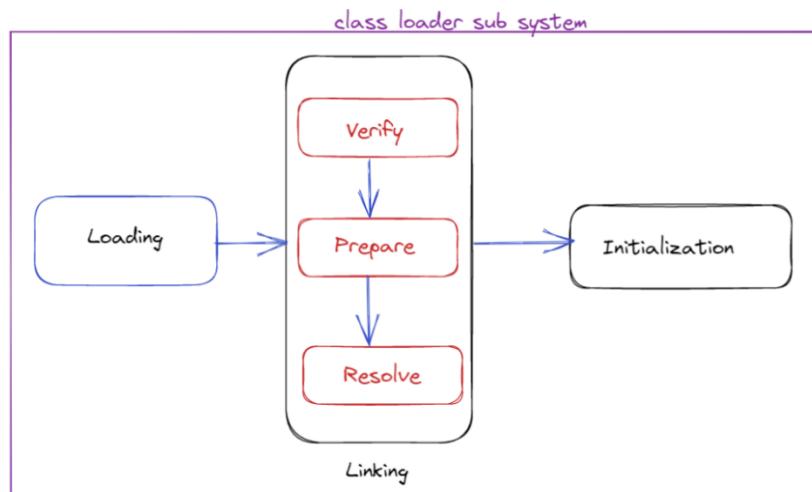


fig: class Loader Process

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 three types of class loaders.

1. Bootstrap class loader / primordial class loader
2. Extension class loader
3. Application class loader / system class loader

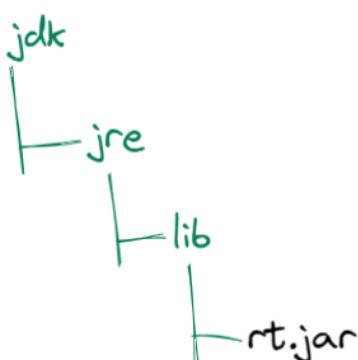
1. Bootstrap class loader / primordial class loader:

Bootstrap class loader is responsible to load core java API classes that is the classes present “ rt.jar ”.

This location is called bootstrap class path that is boot strap class loader is responsible to load class from bootstrap class path.

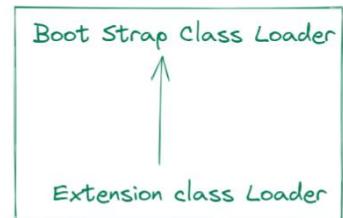
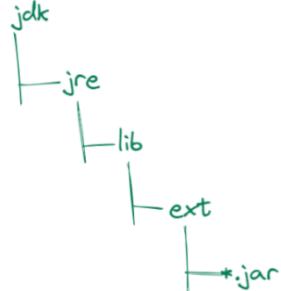
Bootstrap class loader is by default available with every JVM it is implemented in native languages like c or cpp and not implemented in java.

R



2. Extension class loader

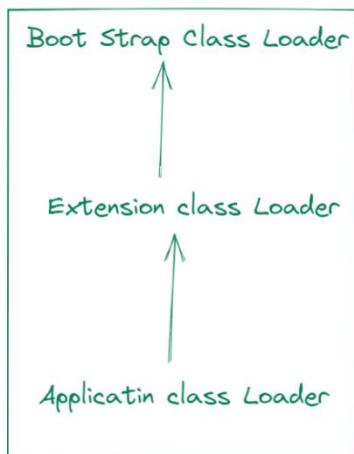
Extension class loader is the child class of Bootstrap class loader.



Extension class loader is responsible to load class from extension class path (JDK\JRE\lib\ext).

Extension class loader is implemented in java and the corresponding .class file is "sun.misc.Launcher\$ExtClassLoader.class".

3. Application class loader / system class Loader

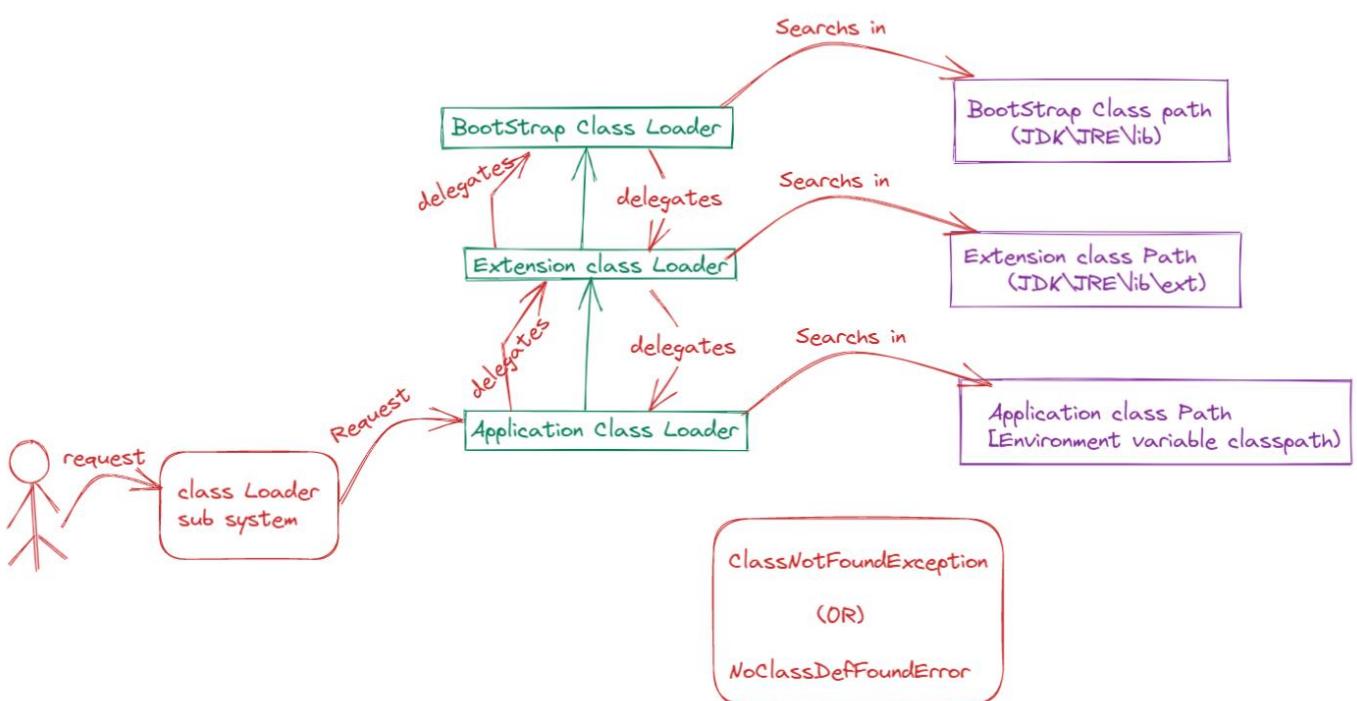


Application class loader is the child class of extension class loader. This class loader is responsible to load class from application class path.

It internally uses environmental variable class path.

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 come 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 handovers the request to application class loader.

Application class loader delegates the request to extension class loader which intern delegate the request to bootstrap class loader.

Then bootstrap class 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”.

```
public class ClassLoaderDemo
{
    public static void main(String[] args)
    {
        System.out.println(String.class.getClassLoader());
        System.out.println(ClassLoaderDemo.class.getClassLoader());
        System.out.println(CustomerOne.class.getClassLoader());
    }
}
```

//assume CustomerOne.class present in both extension and application class paths and ClassLoaderDemo.class present in only application ClassPath.

For String.class

Bootstrap class loader from Bootstrap class path
Output: null

For ClassLoaderDemo.class:

Application class loader from application ClassPath
Output: [sun.misc.Launcher\\$AppClassLoader@1912a56](#)

For CustomerOne.class

Extension class Loader from extension ClassPath
Output: [sun.misc.Launcher\\$ExtClassLoader@1072b90](#)

Note:

1. Bootstrap class loader is not java object hence we got null in the first case but extension and application class loaders are java objects hence we are getting corresponding output for remaining two sop's (classname@hascode_in_hexadecimaform);
2. class loader subsystem will give the highest priority for bootstrap class path and then extension classpath followed by application classpath.

Need of customized class loader

BHUPATHI MALACHI CORE JAVA NOTES

BHUPATHI MALACHI CORE JAVA NOTES

BHUPATHI MALACHI CORE JAVA NOTES