29th Oct 2020

# Core Java

PAGE No.
DATE / /

## * Final, Finalize (), finally ?

→ **Final** : modifier, applicable for classes, methods & variables.

- If a class declared as final then we can't extend that class. (i.e can't create child class)
- If method declared as final then we can't override that method in child class.

- If a variable declared final then it will bcome constant & we can't perform re-assignment for that variables.

→ **Finally** :
finally is a block always associated with try-catch to maintain cleanup code.

eg

```
try{
    //risky code ...}

Catch (X e)
{
    // Handling code....}
finally
{
    //cleanup Code}
```

→ **finalize()** : method which is always invoked by garbage collector just before destroying an obj. to perform cleanup activities.

Scanned by CamScanner

PAGE No.
DATE / /

* finally meant for cleanup activities related to try block. where as finalize () meant for cleanup activities related to obj.

30/10/20

* String & StringBuffer

| | |
|---|---|
| i> String s = new String ("durga"); | StringBuffer sb = new SB ("durga"); |
| ii> s.concat ("software"); | sb.append ("software"); |
| iii> s.o.p(s); → durga | s.o.p(sb); → durga software |

→

i> s → durga

ii> s → durgasoftware

Since we are not storing o/p anywhere; so this will be given to garbage collector.

iii> durga

sb → durga

sb → durgasoftware

durgasoftware.

1. Once we create a String obj. we can't perform any changes in the existing obj.
If we are ~~try~~ trying to perform any changes with those changes a new obj. will be created. this non-changeable nature is nothing but immutability of String obj.

2. Once we create a StringBuffer obj. we can perform any type of changes in the existing obj. this changeable is nothing but mutability of StringBuffer obj.

Scanned by CamScanner

PAGE No.
DATE / /

## * Diff. Betn StringBuffer & StringBuilder.

→ StringBuilder is exactly same as StringBuffer (including methods & constructors) except the foll. differences:

| StringBuffer | StringBuilder. |
|---|---|
| - Every method is Synchronized | - Non-Synchronized. |
| - At a tym 1 thread is allowed to process on StringBuffer. ∴ Thread-safe. | - Not-thread safe. |
| - Relatively performance is low. bcoz every 2nd thread is in waiting mode. | - Relatively performance is high. |
| - Introduced in 1.0 version | - In 1.5 version. |

## * Diff. Betn. [== operator & .equal()]

== operator ↓ reference comparison.

.equal() ↓ Content comparison

eg

```
String s1 = new String("durga");
String s2 = new String("durga");
```

s1 → durga
s2 → durga

```
s.o.p (s1==s2);
```
→ False → bcoz both obj. are pointing 2 diff. obj.

```
s.o.p (s1.equal(s2));
```
→ True
↳ As both obj. are pointing 2 diff. obj.; but Content Stored is same.

Scanned by CamScanner

PAGE No.
DATE / /

Note*

- .equal() method present in obj. class also meant for reference comparison only based on our requirements we can ~~our~~ override for content comparison.

- In String class, all wrapper class & all collection classes .equal() method is overridden for content comparison.

4th Remaining [Start] tommorow)

* String vs StringBuffer vs StringBuilder?

When to use what:

- If the content is fixed & won't change frequently then we should go for String.

- If the content is not fixed & keep on changing but Thread Safety is required then we should go for StringBuffer.

- If the content is not fixed & keep on changing & thread safety is not required then we should go for StringBuilder.

Scanned by CamScanner

1st Nov 2020

PAGE No.
DATE / /

# * Modifiers in Java

| Modifiers | Class I | Class O | Methods | Vari. | Blks | Interfac I | Interfac O | Enum I | Enum O | Constru |
|---|---|---|---|---|---|---|---|---|---|---|
| Public | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ |
| Private | — | ✓ | ✓ | ✓ | | | ✓ | | ✓ | ✓ |
| Protected | | ✓ | ✓ | ✓ | | | ✓ | | ✓ | ✓ |
| default | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ |
| final | ✓ | ✓ | ✓ | ✓ | | | | | | |
| abstract | ✓ | ✓ | ✓ | | | ✓ | ✓ | | | |
| Static | | ✓ | ✓ | ✓ | ✓ | | ✓ | | ✓ | |
| Synchronized | | | ✓ | | ✓ | | | | | |
| Native | | | ✓ | | | | | | | |
| Strictfp | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | |
| Transient | | | | ✓ | | | | | | |
| Volatile | | | | ✓ | | | | | | |

Note*

- The modifiers which are applicable for inner classes but ~~&~~ not for outer classes.

- Modifiers which are applicable for classes but not for interface are <u>final</u>

- Modifiers which are applicable only for methods & which we can't use anywhere else <u>Native</u>

- Interface which is declared inside a class is always Static where we are declaring or not.

Scanned by CamScanner

PAGE No.
DATE / /

* - Interface which is declared inside interface is always public & static whether we are declaring or not.

* - Class which is declared inside interface is always Public & Static whether we are declaring or not.

→ Illegal Combinations:

① Methods

Public — Private, Protected } ×

abstract — final, static, synchronized, native, private, strictfp } ×

② Variables

Public — Private, Protected } ×

Final → Volatile ×

③ Class level

Final → abstract ×

Public — Private, Protected } ×

Scanned by CamScanner

PAGE No.
DATE / /

## * Interface Class Vs Abstract Class vs Concrete Class

i) Interface Class → Servlet → Plan

ii) Abstract → Generic Servlet & HttpServlet → Partially Complete Building

iii) Concrete Class → MyOwnServlet → Fully Completed Building.

**Notes**

- If we don't know anything abt implementatn just we have requirenment specificatn (100% Abstractn) then we should go for interface
- If we are talking about implementatn but not completely (partial) then we should go for Abstract.
- we are talking about implementatn completely & ready to provide service then we should go for Concrete Class

## * Access Specifiers VS Access Modifiers

- In old lang. lyk C++ public, private, protected, default are considered as Access Specifiers. Except this the remaining (lyk static) are considered as Access Modifiers.

Scanned by CamScanner

PAGE No.
DATE / /

- But in Java there is no terminology lyk Specifiers, all are by default considered as Modifiers only

| | | |
|---|---|---|
| Public | static | default |
| Private | Synchronized. | Native |
| Protected | strictfp (1.2v) | |
| final | volatile | |
| abstract | transient. | |

eg

```
private class Test
{
}
```

Compile Error:
modifier private not allowed here.

Outer Class / Top-level
↳ modifiers allowed
Public
<default>
final
abstract
strictfp

9th video

7th Nov 2020

## * Diff. betn Abstract & Interface Class *

| Interface | Abstract Class |
|---|---|
| - If we don't know anything abt implementatn just we have requirement specificatn then we should go for interface. | - If we are talking abt implementatn but not completely (partial implementatn) then we should go for Abstract class. |
| ~~- Inside interface every method~~ | |
| * - interface Test { int x; } CE: equal (=) must be there (initializatn must be there) | - Not such things mandatory. |

Scanned by CamScanner

[4, 5, 6, 7, 8] imp pts

- interface Test
  {
  Test()
  {
  }
  }

→ within interface every variable is static, already initialized; so need of instance variable is not required

→ purpose of constructor is to initialize instance variable

## System.out.println() ?

eg →

```
class Test
{
static String s = "java";
}
```

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

Test.s.length();

- Test → Test is a class name
- s → 's' is a static variable present in Test class, of type String.
- length() → length() is a method present in String class.

System.out.println();

- System → class ~~Class~~ present in java.lang package
- out → Out is static variable present in Sys. class of type PrintStream.
- println() → is a method present in PrintStream class.

Scanned by CamScanner

8th Nov 2020

PAGE No.
DATE / /

# * Overloading vs Overriding :

overloaded → 2 methods are said to be overloaded if & only if both methods havings the same name but diff. arg. types.
arg → argument

eg
```
Class Test{
  public void m1(int i)
  {
  }
  public void m1(long l)
  {
  }
}
```
overloaded method.

→ whatever methods parent has by default available to the child through inheritance. Some times child may not satisfy with parent method implementat^n. Then child is allow to redefine that method based on its requirenment. this process is called overriding.

The parent class method which is overridden is called overridden method.

The child class method which is overriding is called overriding method.

eg
```
Class Parent
{
    Public void property ()
    {
```

Scanned by CamScanner

final

overridden method →

```
S.O.P ( cash , land , Gold ) ; → property
}

public void marry ()
{
    S.O.P ("Sublalaxmi") ; }}

Class child extends parents
{
    pub. void marry()
    {
        S.O.P ( "Sneha | Sneha | mau") ; }
    }}
```

overriding

overriding method →

so changing method defined in parent class.

24th Jan 2020

## PART-II

| Property | Overloading | Overriding |
|---|---|---|
| i) method names | must be same | must be same |
| ii) Argument types | must be diff. (atleast order) | must be same (including order) |
| iii) method signature | must be diff. | must be same. |
| iv) Return type | No restrictions | must be same until (1.4v) only. 1.5v Co-variant. |

Scanned by CamScanner

PAGE No.
DATE / /

* Note
Co-variant return type
[explaination for 4th ~~5th~~ pt]

p: public Object m1()

c: public Object m1()

upto version 1.5 it is must; but after 1.5 version "co-variant" Concept came into picture i.e child class can have return type child class also.

eg
c : public object m1()
String / StringBuffer.

xox

| | | |
|---|---|---|
| ✓ private, static & final methods | Can be overloaded | Can not be overridden. |

Explaination

Private
p:
private void m1()
c:
not available.
→ not possible in java to override private method.
- only visible & accessible in class where they have declared.
- private keyword provides highest level of Encapsulat^n in Java

Static
you can have static methods in child class; but that concept is method hiding not overriding

PAGE No.
DATE / /

final

If class implementatn declared final ; then we can't change anything.

| | | |
|---|---|---|
| ii) Access Modifier | No restrict^n | - we can't reduce scope of Access Modifier but we can increase |
| iii) throws clause | No restrict^n | - If ~~ch~~ child class method throws any checked excep^tn Compulsory parent class method should throw the same checked exception ~~and~~ or its parent otherwise we will get compile tym error but there are no restrictions for unchecked excep^tn. |
| iii) Method Resolution | - Always takes care by Compiler based on reference type | - Always takes care by JVM based on Runtime object. |
| iv) Also known as | Compile time polymorphism or static poly. or early Binding. | - Runtime poly, dynamic poly or late binding. |

* Note - In overloading we have to check only method names (must be same) & argument types (must be diff) except this the remaining like return types, access modifiers etc., are not required to check.

Scanned by CamScanner

Durga S/w

Remaining videos :- 11, 12, 13, 17, 18, 19, 20, 21, 22

- But in overriding every thing we have to check like method names argument types, return type, access modifiers etc.

## * Diff Bet

Parent p = new Child();
&
Child c = new Child(); *

| Child c = new Child(); | Parent p = new Child(); |
|---|---|
| - If we know exact runtime type of obj. then we should use this approach. | - If we don't know exact runtime type of obj. then we should use this approach. (polymorphism). |
| - By using child reference we can call both parent & child class methods. | - By using parent reference we can call only methods available in parent class & child ~~clas~~ specific methods we can't call. |
| - we can use child reference to hold only for that particular child class obj. only. | - we can use parent refernce to hold any child class object. |

Scanned by CamScanner

30th Jan 2021

# Exception Handling

## 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
Statements Executes:
(1, 2, 3, 5)
Normal Terminat$^n$ (NT)

**Case 2:**
If an except$^n$ raised at Statement 2 & Corresponding catch block matched.
1 4, 5, NT

**Case 3:**
If an except$^n$ raised at Statement 2 & corresponding Catch block not matched.
1, Abnormal Terminat$^n$ (AT)

**Case 4:**
If an except$^n$ raised at Statement 4 or Statement 5.
- It is always abnormal termination.

Scanned by CamScanner

PAGE No.
DATE / /

## * Conclusions:-

i) with in the try block if anywhere except raised then rest of try block won't be executed even though we handled that except.
Hence length of try block should be as less as possible. And we have to take only risky code with in try block.

ii) In addition to try blk there may be a chance of raising an except in inside catch & finally blks.

iii) If any statement raises an except, which is not part of try blk then it is always Abnormal Termination.

## * Control-flow *

(try - catch - finally)

```
try {
    Stmnt 1;
    Stmnt 2;
    Stmnt 3; }

Catch (x e) {

    Stmnt 4; }
```

```
finally {

    Stmnt 5; }

Stmnt 6;
```

Scanned by CamScanner

PAGE No.
DATE / /

Case 1:
If there is no exception

1, 2, 3, 5, 6, NT

Case 2:
If an excep^n raised at stmnt 2 & corresponding catch blk matched

1, 4, 5, 6, NT

Case 3:
If an excep^n raised at stmnt 2 & the corresponding catch blk not matched.

~~1, 2, 5, 6, AT~~ 1, 5, AT

Case 4: If an excep^n raised at stmnt-4 then it is always AT but before AT finally blk will be executed.

Case 5:
If an excep^n raised at stmnt-5 or at stmnt-6.
then it is always AT.

Scanned by CamScanner

31st Jan 2021

PAGE No.
DATE / /

# * Various possible Combinations of try - catch - finally Block *

i) whenever we are writing try block compulsory we should write catch or finally. That is 'try' without catch or finally is invalid syntax.

ii) whenever we are writing catch block compulsory we should write try block that is catch without try is invalid.

iii) whenever we are writing finally block; compulsory we should write try block. That is finally without try is invalid.

iv) In try catch finally; order is important.

v) 'Try' with multiple catch blocks is valid but the order is important; compulsory we should take from child to parent.

By mistake if we are trying to take from parent to child then we will get compile time error.

Scanned by CamScanner

ii) If we are defining 2 catch blocks for the same excep^n then will get compile time error.

iii) we can define try-catch-finally with in the try, with in the catch & with in finally blocks.
Hence nesting of try-catch-finally is valid.

iii) For try-catch-finally. curly braces are mandatory.

## Combinations

① 
```
try
{
}
Catch(x e)
{
}
```

②
```
try
{}
Catch(x e)
{}
Catch(x e)
{}
```
CE: excep^n x has already been caught.

③
```
try
{}
Catch (x e)
{}
Catch (y e)
{}
```

④
```
try
{}
Catch (Exception e)
{}
Catch (ArithmeticException e)
{}
```
CE :- Exception java.lang.AE has already been caught.

Scanned by CamScanner

PAGE No.
DATE / /

(5)
```
try
{
}
Catch (AE e)
{}
Catch (Exception e)
{}
```

(6)
```
try
{}
```
CE: try without Catch or finally

(7)
```
Catch(xe)
{}
```
CE: Catch without try.

(8)
```
Finally
{}
```
CE: finally without try

(9)
```
try
{}
finally
{}
```

(10)
```
try
{}
S.O.P ("Hello");
Catch(x e)
{}
```
CE1:- try without Catch or finally
CE2:- Catch without try.

(11)
```
try
{}
Catch(x e)
{}
S.O.P("hello");
Catch(y, e)
{}
```
CE: Catch without try

(12)
```
try{}
Catch(xe)
{}
S.O.P ("hello");
finally
{}
```
CE:- finally without try.

Scanned by CamScanner

PAGE No.
DATE / /

(13)
```
try
{}
Catch(x e)
{}
try
{}
finally
{}
```

(14)
```
try
{}
finally
{}
Catch(x e)
{}
```
CE:- Catch without try.

(15)
```
try
{}
Catch(x e)
{}
finally
{}
finally
{}
```
CE:- finally without try.

(16)
```
try
{
try
{}
Catch(x e)
{}
finally
{}}
Catch(x e)
```

(17)
```
try
{}
Catch(x e)
{
try
{}
finally
{}}
```

(18)
```
try
{}
Catch(x e)
{}
finally
{
try
{}
finally
{}}
```

(19)
```
try
{}
try
{}}
Catch(x e)
{}
```
CE: try without Catch or finally.

Scanned by CamScanner

PAGE No.
DATE / /

(20)
```
try
S.O.P ("hello");
catch (x e)
{
}
```
×

(21)
```
try
{
}
catch (x e)
S.O.P ("catch");
```
×

(22)
```
try
{
}
catch (x e)
{ }
finally
S.O.P ("finally")
```
×

## * Difference Between Exception & Error *

Throwable acts as root ~~of~~ for java exception hierarchy.

- Throwable class contains 2 child classes:

  i) Exception  ii) Error.

Most of the cases exception are caused by our program & this are recoverable.

eg Money withdrawable ; FileNotFound.

ii) Errors

Most of the times errors are not caused by our program these are due to lack of system resources.

- errors are non-recoverable.

Scanned by CamScanner

PAGE No.
DATE / /

eg ⇒ Out of Memory error occurs being a programmer we can't do anything & the program will be terminated abnormally.

- Sys. admin or server admin is responsible to increase heap memory.

## * Diff Between Checked & Un-checked Exception *

- **Checked**

The exceptions which are checked by compiler for smooth execution of the prg. at runtime are called checked exception.

eg ⇒ HallTicketMissingExcept$^n$
PenNotWorkingExcept$^n$
FileNotFoundExcept$^n$.

In the case of checked except$^n$ compiler will check whether we are handling except$^n$ if the programmer not handling then we will get compile tym error.

- **UnChecked**

The except$^n$ which are not checked by compiler are called Unchecked exceptions.

Scanned by CamScanner

PAGE No.
DATE / /

Examples

ArithmeticException
BomBlastExcept$^n$
NullPointerExcept$^n$.

- In the case of Unchecked Except$^n$ Compiler won't check whether programmer handling exception or not.

Note* i) whether exception is checked or Unchecked compulsory it will occure only at Runtime. There is no chance of occuring any exception at compile time.

ii) Runtime exception & its child classes, error & its child classes are unchecked this remaining are checked Except$^n$.

## * Fully Check & Partially Checked *

- A checked exception is said to be fully checked exception if & only if all its child classes also checked.

eg
IOException
InterruptedException

- A checked exception is said to be partially checked Except$^n$ if & only if

Some of its child classes are unchecked.

eg. Exception

*Note - The only possible partially checked exceptions in java are :-

i> Exception
ii> Throwable.

xox

1st feb 2021

## * throws *

| throw | throws |
|---|---|
| - used to explicitly throw an exception. | - used to declare an excep$^n$ |
| - checked Exception cannot be propagated | - Can be propagated |
| - followed By instance | - followed by class |
| - Used within the method | - with the method signature. |
| - You cannot throw multiple excep$^n$ | - you can. |

eg = Public void method()
throws
IOException, SQLException

Scanned by CamScanner

PART-I

PAGE No.
DATE / /

# * Explain about Main Method?

- whether class contains main() or not & whether main() is declared according to requirement or not these things won't be checked by compiler. At runtime JVM is responsible to check these things.

- At runtime if JVM is unable to find required main() method then we will get runtime exception saying
  NoSuchMethodError.main

```
Class Test
{
}
```

javac. Test. java
java Test
RE :- NoSuchMethodError : Main

**Public** → To call by JVM from anywhere

**Static** → without existing obj. also JVM has to call this method & main method no way related to any obj.

**Void** → main() method won't return anything to JVM

**Main(String args[])** → Command line arguments. this is name which is configured inside JVM

PAGE No.
DATE / /

- The above syntax is very strict if we perform any change we will get runtime exception saying NoSuchMethodError: main

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

1. The order of modifiers is not important that is instead of "public static" we can take "static public" also.
2. we can declare "String[]" in any acceptable form.
   main (String [] args)
   main (String [] args)
   main (String args[])
3. instead of "args" we can take any valid java identifier.
4. we can replace "String[]" with var arg parameter
   main (String [] args) ==> main (String ... args)
5. we can declare main() method with the following modifiers also.
   final
   synchronized
   strictfp

Scanned by CamScanner

(6) Class Test
{
final static synchronized strictfp public void main(String... durga)
{
S.open("void main");
}}

Javac Test.java
Java Test
Output: void main

(7) which of the following are valid main method declarations?

a) public static void Main(String[] args)
b) public static int main(String[] args)
c) public static void main(String ...args)
d) public final synchronized strictfp void main(String[] args)
e) public static final synchronized strictfp void main(String args)
f) public static void main(String...args)

static is missing

PART-II

Case 1:

- overloading of the main method is possible but JVM will always call String[] argument main method only.

- The other overloaded method we have to call explicitly then it will be executed just a normal method call.

```
Class Test
{
    public static void main
        (String[] args)
    {
        S.O.p("String[]");
    } overloaded methods

    public static void main
        (int[] args)
    {
        S.O.p("int[]"); }}
```

overloaded methods

→ O/p :- String[]

Case 2 :- Inheritance concept applicable for the main method. Hence while executing child class; if child class doesn't contain main method then parent class main method will be executed.

P.java

```
Class P
{
    p s v m(String[] args)
{
    S.O.p("parent main");
}}
Class C extends P
{ }
```

Scanned by CamScanner

Javac P.java

P.class — C.class

java P
o/p:- parent main

java C
o/p: parent main

**Case3:-** It seems overriding concept applicable for main method but it is not overriding it is method hiding.

```
class P
{
P s v m (String[] args)
{
s.o.p ("parent main");
}}

class c extends P
{
P.s.v.m (String[] args)
{
s.o.p ("child main");
}}
```

It is method hiding but not overriding

**Note**

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

Scanned by CamScanner

PAGE No.
DATE / /

# PART III

## * 1.7v Enhancement wrt Main() Method *

**Case 1 :-**

Until 1.6v if the class doesn't contain main() method then we will get runtime exception saying no such method error. But from 1.7v onwards Instead of NoSuchMethodError we will get more meaningful error information.

| 1.6 version | 1.7 version |
|---|---|
| - javac Test.java | javac Test.java |
| - java Test | java Test |
| RE :- NoSuchMethodError : main | Error : Main method not found in class test, please define main method as p.s.v.m (String[] args) |

**Case 2** From 1.7v onwards to run a java prg main method is mandatory. Hence, even though class contains static blocks they won't be executed if the class doesn't contain main() method.

eg :

```
class Test
{
    static {
        S.O.P ("Static Block"); }}
```

Scanned by CamScanner

| 1.6v | 1.7v |
|---|---|
| - javac Test.java | javac Test.java |
| - java Test | java Test |
| o/p : Static block | Error : main method not found in class. |
| RE :- NoSuchMethodError : main | |

eg2

```
Class Test
{
Static
{
s.o.p ("Static Block");
Sys. exit(0);
}}
```

| 1.6v | 1.7v |
|---|---|
| - javac Test.java | javac Test.java |
| - java Test | java Test |
| o/p :- Static Block | - Error : Main method not found in class |

case3

If the class contains main(); whether it is 1.6v or 1.7v there is no change in execution sequence.

```
Class Test
{
static
{
s.o.p("static block");
}
p.s.v.m(String[] args)
{
s.o.p("main method");
}}
```

| 1.6v | 1.7v |
| --- | --- |
| javac Test.java | javac Test.java |
| java Test | java Test |
| o/p: Static Block<br>Main method | o/p: Static Block<br>Main method. |

→ with out writing main() method is it possible to print some stmts to the Console?

Yes, By using static Block we can do. until version 1.6.

From 1.7v onwards main() method is mandatory to print some stmts to the Console.

## WRAPPER CLASS

Need of wrapper class?

8 primitive datatypes :-

byte
short
int
long
float.
double
char
boolean

As java is Obj.-oriented prg. lang most of tym it requires obj. as i/p.

eg

```
Arraylist l = new Arraylist();
l.add(10);
```

↳ primitive data.

In 1.4v it will give an error.

But in 1.5v if compile this; it will get compiled with the help of Autoboxing Concept.

eg

```
import java.util.*;

Class Test
{
psvm(String[] args)
{

Arraylist l = new Arraylist();
l.add(10); }}
```

→ If in our sys. Higher version is available & we want to run 1.4v code on it :-

javac -source 1.4 Test.java

Error :- no suitable method found for l.add(10);

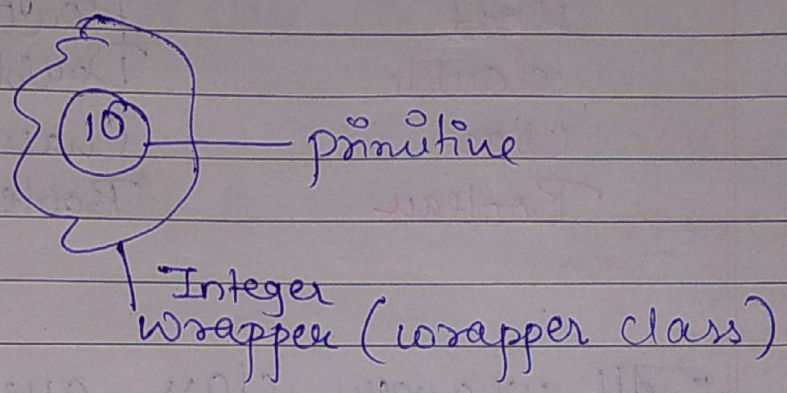
method Collection.add(Object) is not applicable.

Scanned by CamScanner

→ To make it working :-

Convert primitive type into object type.

How to convert primitive to Object ?




```
Integer I = new Integer(10);
l.add(I);
```

→ Convert this primitive value to obj form.

purpose of wrapper class.

i) To wrap primitive into obj. form.

ii) ~~Stud~~ To define several Utility methods for primitives. (inside wrapper class).

- eg Suppose [10] is primitive value & you want to convert this to obj. form

```
String s = Integer.toString(10);
```

- present in Integer class
- Static method.

Scanned by CamScanner

→ How many wrapper class is/are available?

| | |
|---|---|
| Byte → | Byte |
| Short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| Char | Character |
| Boolean | Boolean. |

**Note**

- All wrapper class are final classes.
- All wrapper class are declared in

  java.lang.package.

## * Object Class *

Java.lang.Object

↓

It is a parent class to any class in java.

eg

(1) A.java

```
class A {
}
```

⟹

(2)

```
Class A extends java.lang.Object
{
}
```

↓

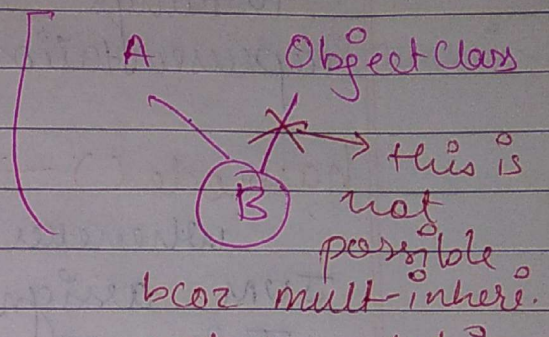
we have not mentioned that this class will act as child class for an Object class

PAGE No.
DATE / /

javac A.java
↓ calls
compiler
(will check whether this class is having parent class or not?
If not then it will convert it to child class. (2))

eg (2) B.java.

```
Class B extends A
{
}
```




Object class
↑
A
↑
B

{ directly or indirectly child class is associated with obj. class. }

→ Purpose of Object Class?

- Contain common methods which are required for any class.
- Contains 11 methods :-

i) toString()
ii) hashCode()
iii) equals()
iv) clone()
v) finalize()
vi) notify()
vii) notifyAll()
viii) wait()
ix) wait(long ms)
x) wait(long ms, int ns)
xi) getClass()

milli-seconds
nano-seconds

Scanned by CamScanner

PAGE No.
DATE / /

toString() → whenever we build any class reference; automatically toString() will be called.

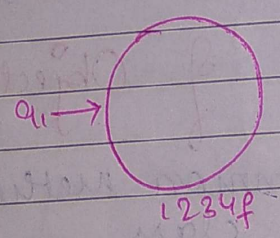
- whenever user wants to print any reference data/information we write that inside toString method.

- toStrings represents obj. form in String representation.

hashCode() →
whenever we create a new object; JVM assigns a unique value. This unique value is called hashCode.

hashCode is generated on the basis of address of obj.; to identify obj. uniquely.

A a1 = new A();




equals() → to compare content of obj.

clone() → when we want exactly duplicate copy of parent obj.

finalize() → when we make any obj eligible for garbage collection.

a1 = null.

3rd Feb 2021

Scanned by CamScanner

Before destroying any obj. garbage Collector always calls finalize().
In finalize() we put cleanup code

getClass() → to get current classname.
getclass() returns class a class's obj.

notify() →
notifyAll() →
wait() →
wait(long ms)
wait(long ms, int ns) →
} Used for inter-thread communication.

3rd Feb 2021

## * Singleton class *

Class which is allowed us to create only one obj. is nothing but a singleton class.

- class restricts the unnessary creation of an obj. whenever a new request is made it uses the existing obj. instead of creating new obj. which results in no memory wastage.

→ How to Create?

i) declare static data member which is a instance of singleton class.

Scanned by CamScanner

ii) Create private Constructor which restrict to create instance of singleton class.

iii) Create static factory method which allows to create & access the obj.

eg

```
public static ContactDAO {
private static ContactDAO getInstance()
{ if (contactDAO == null)
  {contactDAO = new ContactDAO();
  }
return contactDAO;
}

~~P.s.v.m (String[] args){~~
public void getContactDetails()
{s.o.p ("Contact Details");}

p.s.v.m (String[] args)
{contactDAO obj1 = ContactDAO.getInstance();
s.o.p (obj1);
}}
```

Scanned by CamScanner

## * Why pointers are not used in Java?

→ - Bcoz they are unsafe & increases the complexity of the prg.

- Since Java is known for its simplicity of code, adding the concept of ptr's will be contradicting.

- Since JVM is responsible for implicit memory allocation, thus in order to avoid direct access to memory by the user, pointers are discouraged in Java.

Scanned by CamScanner

9th Feb 2021

# COLLECTION FRAMEWORK

PAGE No.
DATE / /

## • Need of Collection :-

An array is an indexed collection of fixed no of homogenous data elements.

The main advantage of Array is we can represent multiple values with a single variable.

So that reusability of code will be improved.

### Limitations of Arrays :-

i) Arrays are fixed in size i.e once we created an array with some size there is no chance of increasing or decreasing it's size based on our requirement.

Hence to use array compulsory we should know the size in advance which may not possible always.

ii) Arrays can hold only homogenous data elts

eg
```
Student [ ] s = new Student [10000];
s[0] = new Student ;  ✓
s[1] = new Customer();  ✗
```

iii) Array concept is not implemented based on some std DS hence readymade method support is not available for every requirement we have to write the code explicitly.

which is complexity of Programming.

Scanned by CamScanner

PAGE No.
DATE / /

To overcome the above limitations of arrays we should go for Collections:-

i) Collections are growable in nature.
ii) Collections can hold both homogenous & heterogenous.
iii) Every Collection class is implemented based on some std DS.

→ Diff Betn Arrays & Collections:-

| Arrays | Collections |
|---|---|
| - Arrays are fixed in size. | - Growable in nature. |
| - wrt memory arrays are not recommended. | - wrt to memory Collectn are recommended |
| - wrt performance Arrays are recommended. | - not recommended. |
| - Arrays can hold only homogenous datatype elts. | - Homogenous & Heterogenous elts. |
| - There is no underlying DS for arrays. | - Every Collections Class is implemented based on some std DS. |
| - Array can hold both primitives & Object types. | - Collections can hold only obj. but not primitives. |

Note

PAGE No.
DATE / /

## What is Collection?

If we want to represent a group of individual objects as a single entity then we should go for collection.

## Collection Framework :-

It defines several classes & interfaces which can be used a group of objects as single entity.

| Java | C++ |
|---|---|
| - Collection | - Container |
| - Collection Framework | - STL (Standard Template Library) ↓ grp of classes & interfaces |

*Note* Interface :- place where we get detailed info / nuclear info / concepts. (total info is available)

Class is dummy thing which implements interface.

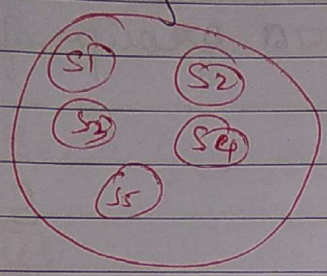
## 4 Key Interfaces of Collection Framework (9 Key interfaces)

1. Collection (I)

- If we want to represent a group of individual obj. as a single entity then we should go for Collection.

- Collection interface defines the most common methods which are applicable for any Collection obj.

eg =




all students obj. will have add(), delete() etc.

- In general Collection interface is considered as root interface of Collect^n framework.

Note 1. There is no concrete class which implements Collection interface directly.

(3) 2. Collection vs Collections

| Collection | Collections |
| --- | --- |
| - Collection is an interface which can be used to represent a grp of individual obj. as a single entity. | - Collections is an ~~entity~~ utility class present in "java.util.package" to define several utility methods. Like sorting, searching for Collection ~~Obj.~~ |

eg =

```
Collections.sort(l);
```

I → interface

(6)

## (2) List (I) :-

- It is a child interface of Collection.

- If we want to represent a grp of individual obj. as a single entity where duplicates are allowed & insertion order preserved then we should go for List.

Collection (I) (1.2V)
↑
List (I)

Implementation classes for List:
- ArrayList (1.2V)
- LinkedList (1.2V)
- Vector (1.0V) → Stack

Legacy classes (classes which are coming from old versions).

(7)

## (3) Set (I) :-

- It is the child interface of Collection.

- If we want to represent a group of individual obj. as a single entity; where duplicates are not allowed & insertion order not preserved then we should go for set.

Collection (I) (1.2V)
↑
Set (I) (1.2V)
↑
HashSet (1.2V)
↑
LinkedHashSet (1.4V)

Scanned by CamScanner

indiui → individual
grp → group
obj → objects

(8)

→ Diff Betn

| List | Set |
|---|---|
| - duplicates are allowed. | - duplicates are not allowed. |
| - Insertion order preserved (in which order data is taken; need to be inserted in that order only) | - not preserved. |

(9) (4) Sorted Set (I) :- (1.2v)
- child interface of set.

- If we want to represent a grp of indiui obj as a single entity where duplicates are not allowed but all obj. should be inserted according to some sorting order then we should go for Sorted Set.

(5) Navigable Set (I) : (1.6v)
It is the child interface of SortedSet it defines several methods for navigation purposes.

* Treeset modified in (1.6v) to implement NavigableSet.

Collection (I) (1.2v)

Set (I) (1.2v)

SortedSet (I) (1.2v)

NavigableSet (I) (1.6v)

TreeSet (1.2v)

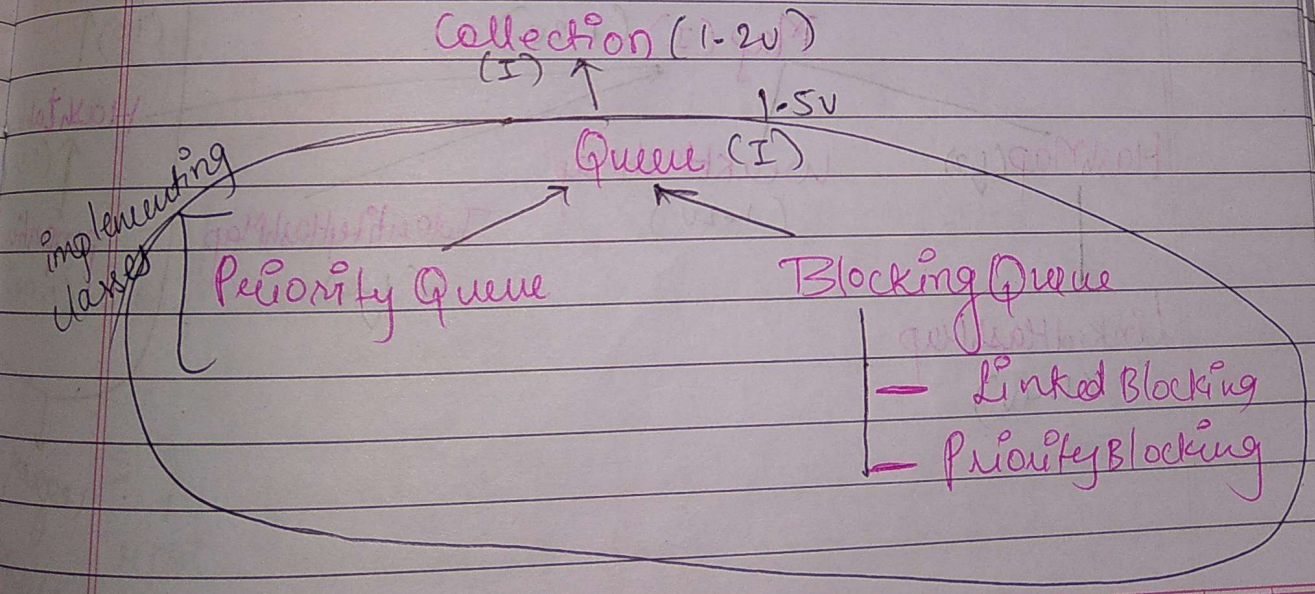
(10)

## 6) Queue (I) :- (1.5)

- child interface of collection.
- If we want to represent a grp of indiv. obj. prior to processing then we should go for Queue.

eg

Before sending mails ; all id's we have to store somewhere & in which order we saved, in the same order mail's should be delivered (FIFO) for this requirement Queue Concept is the best choice.



**Note** All above 6 interfaces ; meant for representing a group of individual obj.

- If we want to represent a grp of obj. as key-value pairs then we should go for Map interface.

## 7 Map (I) :-

It is not child interface of collect.

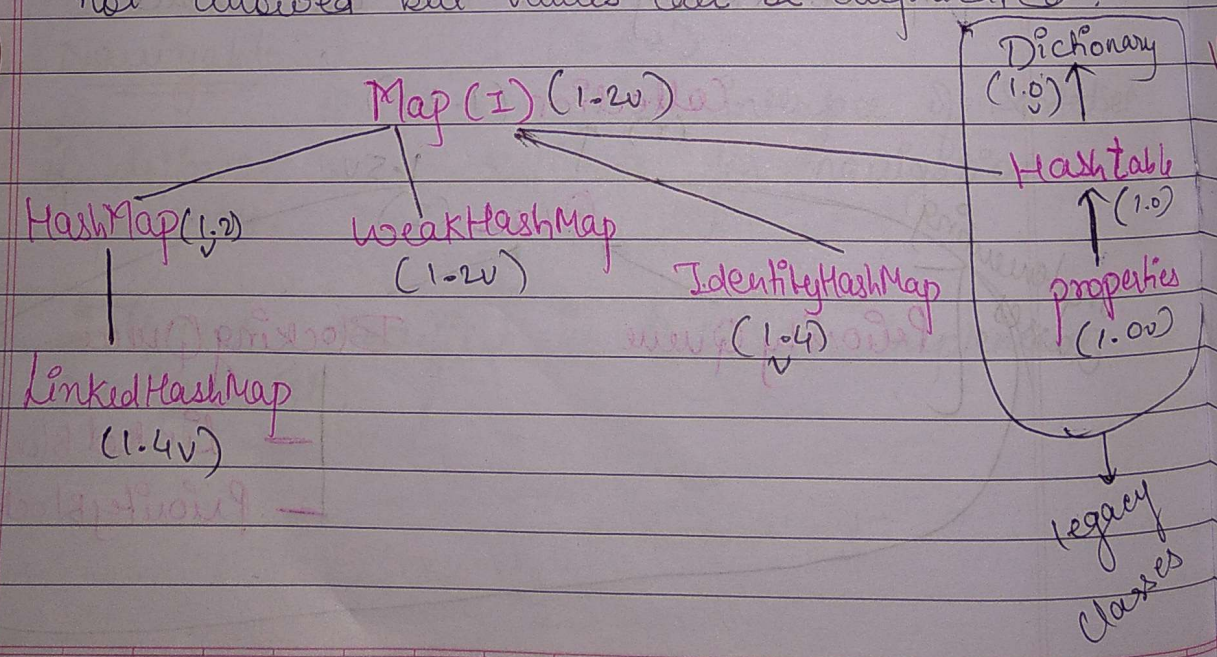
- If we want to represent a grp of indiv. obj. as key-value pairs then should go for Map.

eg

| RollNo | Name |
|---|---|
| 101 | Durga |
| 102 | Rani |
| 103 | Venkat |

Both key & value are obj, duplicated keys are not allowed but values can be duplicated.

10/2/21



(12)

## (8) SortedMap (I) :-

- 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.

Map (I) (1.2v)
└ SortedMap (I) (1.2v)

## (9) NavigableMap (I) :-

It is the child interface of sorted map, it defines several utility methods for navigation purpose.

Map (I) (1.2v)
└ SortedMap (I) (1.2v)
└ NavigableMap (I) (1.6v)
└ TreeMap (1.2v)

10/2/21

(14)

## • Collection Interface :-

- If we want to represent a grp of indiv obj. as a single entity then we should go for Collect$^n$.

- In general Collect$^n$ interface is Considered as root interface of Collect$^n$ framework.

Scanned by CamScanner

- Collectn interface defines the most common methods which are applicable for any Collectn obj.

## * Imp methods of Collection Interface :-

boolean add (object o) → to add single obj
boolean addAll (collectn c) → to add grp of obj.

— " — remove (obj. o)
— " — removeAll (collectn c)
— " — retainAll (Collectn c) → to remove all obj. except those present in c.
void clear ()
boolean contains (obj. o)
— " — ContainsAll (collectn c)
— " — isEmpty ()
int size ()
object[] toArray () → Converting obj. to array
Iterator iterator () → to get my obj. one by one.

**Note*** Collectn interface doesn't contain any method to retrieve obj.; there is no concrete class which implements collectn class directly.

(15)

**List (I) :-** child interface of collectn.
- If we want to present a grp of individual obj. as a single entity where duplicates are allowed

& insertn order must be preserved then we should go for List.

- we can differentiate duplicates by using index.

- we can preserve insertion order by using index, hence index play very imp. role in list interface.

10/2/21

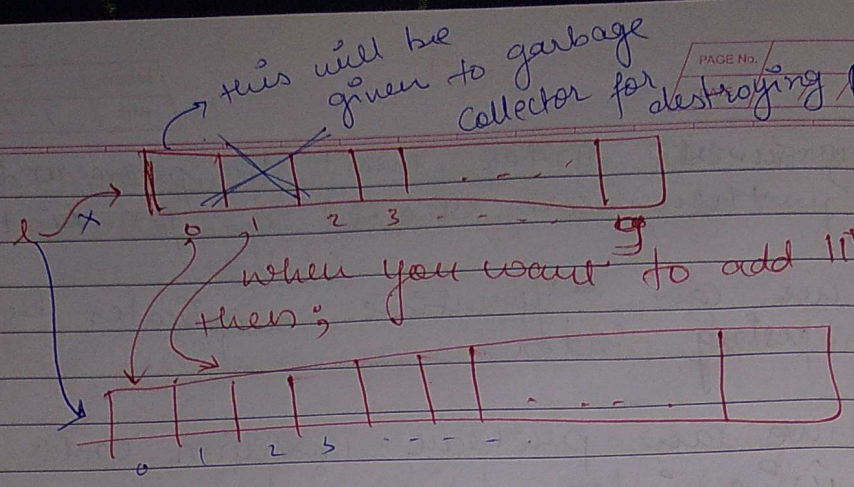
## * ArrayList *

- Underlined DS resizable Array or Growable Array.
- Duplicates are allowed.
- Insertion order is preserved.
- Heterogenous obj. are allowed [except Treeset & TreeMap]
- Null insertion is possible.

→ Constructor in ArrayList :-

(1) ArrayList al = new ArrayList()
Creates an empty Array List obj. with default initial capacity 10.
- Once ArrayList reaches its map capacity a new Array List will be created with

$$\text{new Capacity} = \left(\text{current Capacity} * \frac{3}{2}\right) + 1$$




② Arraylist l = new Arraylist (int initialcapacity);

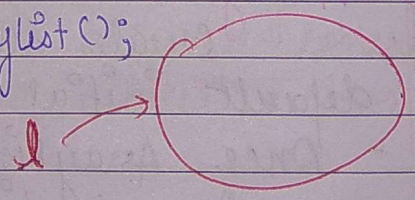
③ Arraylist l = new Arraylist (collection c)
mostly used when inter-change/ Conversation is going to take place.

eg
Linkedlist → Arraylist
TreeSet → Arraylist

eg
```
import java.util.*;
Class ArrayListDemo {

P s v m (String[] args){

Arraylist l= new Arraylist();
l.add ("A");
l.add (10);
l.add ("A");
l.add (null);
S.O.p(l); ⟶ // [A,10,A,null].
l.remove (2);
S.O.p (l); ⟶ // [A,10,null].
l.add ("2","m");
```




S.O.p(l) ⟷ S.O.p (l.toString())

```
l.add ("n
S.O.p (l
}}
```

* Note - Serizable

a copy
process
over-ll
send

easy. It

- Usually
transfe
place
emere
serial

- Every
Serial

- Array
Acces
Rando

- Hence
Ope

```
l.add ("N");
S.O.P (l); ---> // [A, 10, M, null, N]
}}
```

* Note - Serizable Interface

used when you need to store a copy of the obj, send them to another process which runs on the same system or over the n/w. Bcoz you want to store or send an obj.

It makes storing & sending obj. easy. It has nothing to do with security.

- Usually we can use Collect^n to hold & transfer obj. from one place to another place, to provide support for this requirement every collect^n already implements ~~seriza~~ serializable & Clonable interfaces.

- Every Collect^n class implements Both Serializable & Clonable interfaces.

- ArrayList & Vector classes implements Random-Access interface so that we can access any Random elt with the same speed.

- Hence if our frequent operat^n is retrieval Operat^n then ArrayList is the best choice.

## RandomAccess

- present in java.util package.

- It doesn't contain any methods & it is a Marker interface.

eg

```
ArrayList l1 = new ArrayList();
LinkedList l2 = new LinkedList();
S.O.P(l1 instanceOf Serializable);   //true
 -"- (l2 -"- Clonable);              //true

 -"- (l1 -"- RandomAccess);          //true
 -"- (l2 -"- RandomAccess);          //false
```

→ Where to use?

i) ArrayList is the best choice if our frequent operat^n is retrival operat^n
(Bcoz it implements RandomAccess interface)

ii) worst choice if our frequent operat^n is insertion or deletion in the middle.
(Bcoz several shift operat^n are require).

Scanned by CamScanner

(17)

## ArrayList (AL) vs Vector (V)

| ArrayList (AL) | Vector (V) |
|---|---|
| - Query method present in ArrayList is non-synchronize. | - Synchronize. |
| - At a tym multiple threads are allowed to operate on AL obj. & hence AL is not thread safe. | - At a tym only one thread is allowed to operate on V obj. is thread safe. |
| - Threads are not required to wait to operate on AL, hence relatively performance is high. | - Performance is low. |
| - 1.2v & it is non-legacy class. | - 1.0v & it is a legacy class. |

(18)

## How to get Synchronized version of AL obj?

By default AL is obj. is non-synchro. but we can get synchronized version of AL by using Collect^n class synchronizedList() method.

public static List SynchronizedList(List l)

Non-synchronized

```
AL l1 = new AL();
```

Synchronized

```
List l = Collections.synchronizedList(l1);
```

Scanned by CamScanner

- Similarly we can get synchronized versions of Set, Map obj. by using the following methods of Collections class.

public static Set synchronizedSet(Set s);

public static Set synchronizedMap(Map m);

(19)

## Linked List *

- Underlying DS is DLL.
- Insertion order is preserved.
- Duplicates are allowed.
- Heterogenous obj. are allowed.
- Null insertion is possible.
- LL implements Serializable & Clonable interfaces but not RandomAccess interface.

→ Where to use?

- Best choice if our frequent operat^n is insert^n ~~&~~ or deletion in the middle.

- Worst choice if our frequent operat^n is retrieval operat^n.

- Usually we can use LL to implement stacks & queues to provide support for this requirement. LL class defines foll. specific methods:-

void addFirst();
Object getFirst();
Object removeFirst();

void addLast();
Obj. getLast();
Object removeLast();

## LL Constructors :-

① LinkedList l = new LinkedList();
creates an empty LL Obj.

② LL l1 = new LL(Collection c);
Creates an equivalent LL Obj. for the given Collection.

## ArrayList vs LinkedList

| ArrayList | LinkedList |
|---|---|
| - Best choice if our frequent operat$^n$ is retrieval. | - Best choice for Insert$^n$ & Delet$^n$. |
| - Worst choice for insert$^n$ & deletion operation. | - Worst choice for retrieval operat$^n$. |
| - Underlying DS is Resizable / growable array. | - DLL |
| - Implements RandomAccess interface. | - Doesn't implement. |

# * Vector *

- underlying DS for vector is resizable array or growable array.

- Duplicates obj. are allowed.
- Insertion order is preserved.
- 'null' insert$^n$ is possible.
- Heterogenous obj. are allowed.

- Vector class implemented Serializable, Cloneable & RandomAccess Interfaces.

- Most of methods present in Vector are Synchronized Hence vector obj. is thread-safe.

- Best choice if the frequent operat$^n$ is retrival.

## → Vector Methods :-

① For adding objects :-

add ( Object o) → from Collect$^n$ - List(I)
add (int index, Object o) → from list
addElement (Object o) → from Vector.

② For Removing Objects :-

Remove ( Object o) — from Collection
removeElement ( Object o) — from Vector
remove (int index) — from List
RemoveElementAt (int index) — from vector

clear ()
removeAllElements() — from Collect^n / from vector

(3) for Accessing Elements :-

Object get (int index) → from Collection

Object elementAt (int index) → from vector

Object firstElement () → from vector

Object lastElement () → from vector.

(4) Other Methods :-

int size (); → ~~total~~ Current How many obj. are there.

int capacity (); → How many can be fitted.

Enumeration elements (); → when you want obj. one by one from vector.

→ Vector Class Constructor :-

(1) Vector v = new Vector ();

- creates an empty vector obj. with default initial capacity 10, Once vector reaches its max capacity a new vector obj. will be created with

new capacity = 2 * current capacity

(2) Vector v = new Vector (int initialCapacity);

creates an empty Vector obj. with specified initial capacity.

Scanned by CamScanner

But in this ; memory wastage problem is there.

(3) Vector v = new Vector(int initialCapacity, int incrementalCapacity);

(4) Vector v = new Vector(Collection c);
creates an equivalent Vector obj. for the given Collection.

eg

```
import java.util.*;

Class VectorDemo{
p s v m(String arg[]){

  Vector v = new Vector();
  S.o.p(v.capacity());  ---> // 10
  for(int i=0; i<10; i++){

      v.addElement(i);
  }
  S.o.p(v.capacity());  ---> // 10
  v.addElement("A");
  S.o.p(v.capacity());  ---> // 20
  S.o.p(v);  ---> // [0,1,2,3,4,5,6,7,8,9,A]
}}
```

PAGE No.
DATE / /

# STACK *

- child class of vector.
- specially designed class for LIFO

## → Constructor

```
Stack S = new Stack();
```

## → Methods :-

① Object push(Object obj);
for inserting an obj. to the stack.

② Object pop();
To removes & returns top of stack.

③ Object peek();
To returns the top of stack without removal of obj.

④ int search(Object obj);
- If the specified obj is available it returns its ~~ob~~ offset from top of stack.
- If the obj. is not available then it returns -1.

eg

```
S.search("A");
→ 3
S.search("Z");
→ -1
```

| | Offset |
|---|---|
| C | 1 |
| B | 2 |
| A | 3 |

Scanned by CamScanner

(5) Empty();
To check whether the stack is empty or not.

## * Cursors in Java

| Properties | Enumeratⁿ | Iterator | List Iterator |
|---|---|---|---|
| 1) Applicable for | Only legacy classes | Any Collectⁿ class | Only List Classes. |
| 2) Movement | forward | forward | Bidirectional |
| 3) Accessibility | Only Read Access. | Both Read & Remove | Read, Remove, Replace & additⁿ of new obj. |
| 4) How to get it? | By using elements() of vector class. | By iterator() of Collectⁿ interface | Using listIterator() of List interface. |
| 5) Methods | 2 methods<br>- hasMoreElements()<br>- nextElement() | 3 methods<br>- hasNext()<br>- next()<br>- remove() | 9 methods |
| 6) Is it legacy? | Yes (1.0v) | No (1.2v) | No (1.2v) |

Scanned by CamScanner

PAGE No.
DATE / /

| Comparable | vs | Comparator |
|---|---|---|
| - It is meant for default natural sorting order. | | - It is meant for customized sorting order. |
| - Present in java.lang package. | | - Present in java.util package. |
| - This interface defines only one method compareTo(). | | - This interface defines 2 methods compare() equals() |
| - All wrapper classes & String class implements this interface | | - Only implemented classes are Collator RuleBasedCollator. |
| - want to sort array of obj. of same type then we should use this. | | - Sort of array of obj. of diff types. |

Scanned by CamScanner