```
@Override
public void methodOne(){
    ---
}
@Override
public void methodTwo(){
}
}
```

(java.sql.*)
1. statement
2. prepared Statement
3. CallableStatement
4. Connection
5. ResultSet.

Implementation for (java.sql.*) for the body abstact.

Case 3 : An interface can extend any no of interfaces at a time.

eg :
```
interface One {
    public void methodOne();
}
interface Two {
    public void methodTwo();
}
Interface Three extends One, Two {
    public void methodOne();
    public void methodTwo();
    public void methodThree();
}
```

~~Case 4 :~~

public : To make the method available for every implementation class.

abstract : Implementation class is responsible for providing the implementation.

eg: jdbc api (java.sql.*)

implementation should be given by
a. mysql
b. oracle
c. postgresql.

How many access modifiers in java?

a. public, b. private, c. protected, d. static, e. synchronized
f. strictfp, g. final, h. abstract, i. native, j. transient, k
k. volatile. (j and k only for variables)

default ≈ is by default.

Since the methods present inside the interface is
→ public, abstract we can't use the modifiers like
static, private, protected, strictfp, synchronized, native, final.

static ⇒ associated with implementation.
abstract ⇒ can't have implementation.

* Interface Variables.

⇒ Inside the interface we can define variables.
⇒ Inside the interface variables define requirement level constants
⇒ Every variable present inside the interface is by default public static final.

eg:
```
interface ISample {
    int x=10;
}
```

xxx
① notebook Completed refer note book ② to be continued

public :: To make it availble for implementation class Object

Static :: To access it without using implementation class Name.

final :: Implementation class can access the value without any modification.

Why methods are availble without static but variable are made static.

a. Complete information

Method inside interface.

abstract (incompleteness)

Static : complete information should be provided.

```
class Test {
    static void m1(){
    }
```

Variables inside interface are public static final. other access modifiers are illegal combination

a. private b. protected (illegal)
c. transient d. volatile

eg:
```
interface IRemote
{
    int x; EE
}
```

~~Since the variable defined in interface is public static final, we cannot use modifiers like private~~

Since the variable is static and final, compulsory it should be initialized at the time of declaration otherwise it would result in compile time error.

# Keywords

Java keywords are predefined word by java So they Cannot be used as Variable or object name or Class name

Start with small letter

## Reserved words for data types (8)

1. byte
2. Short
3. int
4. long
5. float
6. double
7. char
8. boolean

## Flow control (11)

1) if
2) else
3) switch
4) case
5) default
6) for
7) do
8) while
9) break
10) Continue
11) return

## Exception handling

1) try
2) catch
3) finally
4) throw
5) throws
6) assert (1.4 Version)

## modifiers (11)

1) public
2) private
3) protected
4) static
5) final
6) abstract
7) Synchronized
8) native
9) strictfp (1.2 version)
10) transient
11) volatile

## Class

1) Class
2) package
3) import
4) extends
5) implements
6) interface

## Objects

1) new
2) instanceof
3) Super
4) this

## Reserved (53)

### Keyword (50)

- used (48)
- unused (2) (goto, const.)

### Boolean literal (3)

- true, false } boolean
- null

default value to object reference

ASCII Value :-

32 Space → 64 @ — special Charaters

65 A — 90 Z — Capital Alphabets

97 - a — 122 z — small letter Alphabets

Imp- ASCII Value

⑥ 97 - 65 = 32 — difference we have

⑧ q - 32
97 - 32
= 65 - (A)

(A) ~~A + 32~~
~~A + 32~~
Char. (A + 32)
(q + 32)
65 + 32
(97) ⇒ a ⑧

→ code [b/w local and static final variable in interface always local wins]

```
interface Isample {
    int a=10; // public static final by default
}
public class TestApp implements Isample {
    public static void main (String[] args) {
        int a=20; // local Variable
        System.out.println(a); // output is 20
    }
}
```

(method area is classdata later on jdk 8 version heap area)

(Servelt + x (code)

→ Can a java class implements 2 interfaces Simultaneously?

Yes, possible, but in both the interfaces containing method signature should be same, but different return types.

(compiler should only take method signature)

⇒ interface variables can be accessed from implementation class, but cannot modify if we try to modify it would result in compile time error

eg:

```
interface Remote {
    int VOLUME =100;
}
```

```
class Lg implements Remote {
    public static void main (String... args){
        int VOLUME =0; // local variable
        System.out.println ("Value of volume is::"+VOLUME);
    }
}
```
//o

## Interface Naming Conflicts

### Case 1.

if 2 interfaces contain a method with same signature and same return type in one method implementation is enough.

eg::
```
interface left {
    public void methodOne();
}
interface Right {
    public void methodOne();
}
class Test implements left, Right {
@Override
    public void methodOne (){
        ...
    }
}
```

### Case 2

if 2 interfaces contain a method with same name but different arguments in the implementation for both methods and these methods acts as a overload methods.

Eg:- interface left {
    public void methodOne();
}
interface Right {
    public void methodOne();
}
Class Test implements Left, Right {
    @ override
    public void methodOne(){
        ....
    }
    @override
    public void methodone(int i){
        ...
    }
}

Case 3::

if two interfaces contains a method with same signature but different return types then it is not possible to implement both interface simultaneously.

Eg:- interface Left {
    public void methodone();
}
interface Right {
    public void methodOne(); }
Class Test implements Left, Right{
    @override
    public void methodOne(){

```
@override
  public int methodOne(){
    .....
  }
}.
```

Note :-

Q) Can a java class implements 2 interfaces simultion
yes, possible, expect if two interfaces Contains a method with same signature but different return types.

Variable naming conflicts. ::

Two Variables can contain a Variable with same name and there may be a chance, Variable naming Variable naming

Conficts but we can resolve Variable naming conflicts by using interface names

Example 1 :-

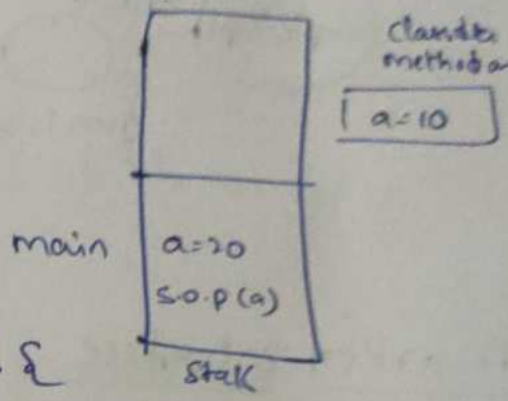
```
interface Left {
    int x=888;
}
interface Right {
    int x=999;
}
public class test implements left, Right {
    public static void main(String...args){
        S.o.p(left.x);
        S.o.p(Right.x);
    }}
```

Note : inside interface the methods are by default "public and abstract".

inside interface the variables are by default "public static and final.

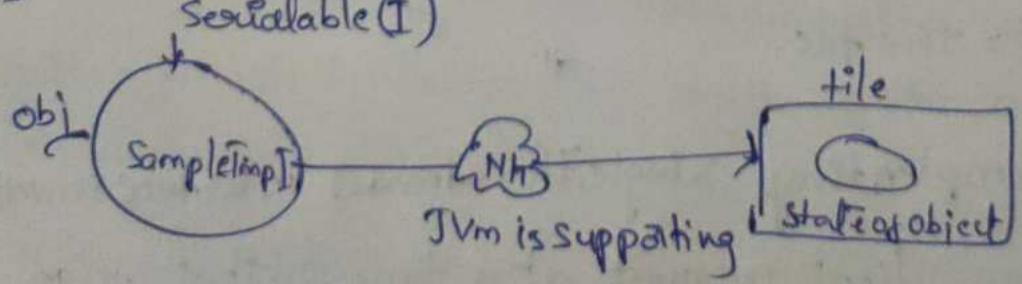
We can also write an interface without any variables & abstract method.

```
interface ISample {
    int a=10; // public static final
}
public class TestApp implements ISample {
    public static void main (String[] args) {
        int a = 20; // local variable
        System.out.println (a);
    }
}
```

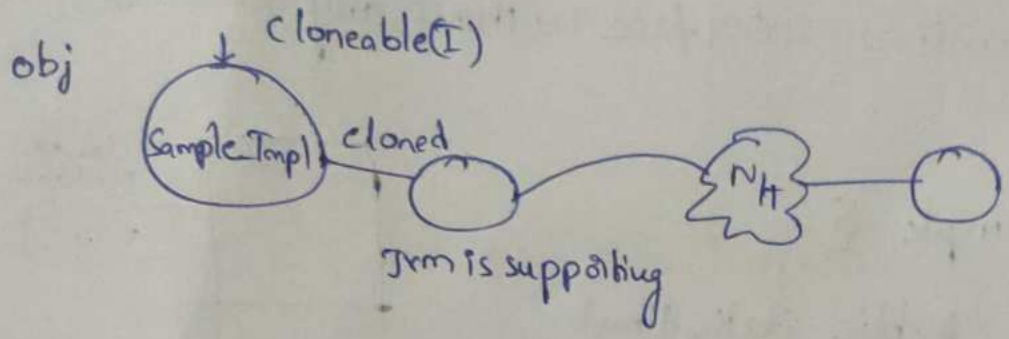



## Marker Interfaces

```
interface Serializable {

class SampleImpl implements Serializable {
}
}
```



```
interface Cloneable {
class SampleImpl implements Cloneable {
}
}
```




## Marker interface

→ if an interface does not contain any methods and methods and by implementing that interface if our Object will get some ability such type of interface are called "Marker Interface" / "Tag Interface" / "Ability Interface".

⇒ Example

Serialazable, cloneable, SingleThreadmodel

Example 1

By implementing Serializable interface we can send that abject accross the network and we can save state of an object into the file.

Example 2 By implenting SingleThreadmodel interface servelt can process only one client request at a time so that we can get "Thread Safety".

example 3 By implemting Cloneable Interface our object is in a (position ~~process only one client~~) position to provide exactly duplicate Cloned object.

without having any methods in marker interface how objects will get ability?

Ans:- JVM is responsible to provide required ability.

Why JVM is providing the required ability to marker Interfaces?

Ans. to reduce the complexity of the programming

Can we create our own marker interface?

Yes, it is possible but we need to customize JVM. (hard ~~for beginners~~)

Adapter Class (It is a design patter allowed to Solve the problem of direct implementation of interface methods).

it is a simple java class that implenments an interface only with empty implementation ~~for~~ every method.

if we implement an interface compulsorily we should give the body for all the methods wheather it is required or not.

This approach increases the length of the code and reduces readability.

eg:-
```
interface X{
    Void m1();
    Void m2();
    Void m3();
    Void m4();
    Void m5();
}
```

```
class Test implements x{
    public void m3(){
        System.out.println("I am from m3()");
    }
    public void m2(){}
    public void m3(){}
    public void m4(){}
    public void m5(){}
}
```

In the above approach, even though we want only m3(), still we need to give body for all the abstract methods. which increase the length of the code. to reduce that we need to use "Adapter class".

Instead of implementing the interface directly we opt for "Adapter class".

Adapter class are such classes which implements the interface and gives dummy implementation for all the abstract methods of interfaces.

So. if we extends adapter classes. then we can easily give body only for those methods which are interested. by giving the body.

eg::

```
Interface x {
    Void m1();
    Void m2();
    Void m3();
    Void m4();
    Void m5();
}
abstract class Adapter x implements x{
    public Void m1(){}
    public Void m2(){}
    public void m3(){}
    public void m4(){}
    public void m5(){}
}
Class TestApp extends Adapter x{
    public void m3(){
        System.out.println("Iam from m3()");
    }
}
```

---

eg!!

Servelet (I)

| implements

GenericsServlet (abstract class)

| extends

HttpServlet (abstract class)

| extends.

myServlet Class

24/11/22

# interface & wrapper class

interface completion

wrapper class

imp topic and interviews questions.

when to go interface, abstract class and concrete class?

interface : it is preferred when we speak only about Specification (no implementation).

abstract class : It is prefered when we speak about partial implementation.

concreate class : it is prefered when we speak about complete implementation and ready to provide service then we go for concrete class.

Difference b/w interface and abstract class?

Interface : if we don't know anything about implementation just we have requirement specification then we should go for interface.

Abstract class : If we are talking about implement ation but not completely then we should go for abstact class.

Interface : Every method present inside the interface is always public and abstact whether we are declaring or not.

abstract class : Every method present inside abstract class need not be public and abstract.

Interface : we can't declare interface methods with the modifiers like private, protected, final, static, Synchronized, native, strictfp.

Abstract : There are not restrictions on abstract class method modifiers

Interface : Every interface variable is always public static final whether we are declaring or not

Abstract : No need to declare anything.

Interface : every interface variable is always public static final we can't declare with modifiers like protected, private ....

Abstract : No restriction on access modifiers

interface : for every interface variable Compulsory we should perform initialization at time of declaration other wise we get Compile time error.

Abstract : Not required to perform initialization for abstract class variables at time of declaration

Interface : Inside interface we can't write static and instance blocks.

Abstract : Inside abstract class we can write static and instance blocks

Interface :- we can't write constructor.

Abstract :- inside abstract we can write constructor.

Note :-

Static block :- .Class file loading happens and used to initialize static Variables.

instance block :- during creating of an object just before the constructor call used for initialization instance Variable.

Constructor :- during creation of an object used for initialisation instance Variable.

During child class object creation, only child class object will be created but no parent class object ~~will be created b~~ (still constructor of parent is called to bring the properties of parent to child.

to get address of object we have to use

S.o.p ln (this.hashcode());

↳ method to print hashcode of object.

Q) Can abstract class be object be created?

A) No

Q) can abstract class contains constructor?

A) Yes.

What is need of abstract class? Can we create an object of abstract class, Does it contain constructor?

→ Abstract class object cannot be created because it is abstract.

→ But constructor is need for Constructor to initialize the object.

Ex:

```
class parent {
    parent () {
        s.o.pln ("parent Constructor");
        s.o.pln (this.hashCode());
    }
}
class child extends parent
    child () {
        s.o.pln ("child Constructor");
        s.o.pln (this.hashcode ());
    }
}
class main() {
    Child c = child ();
    s.o.p (c.hashcode ());
```

---

It will print same hashcode output all

Why abstract Class can contain Constructor where as interface does not contain Constructor?

Abstract Class ⇒ Constructor it is used to perform Initialization of object.

* It is used to provide value for instance Variable.
* It is used to contain instance Variable which are required for child.
* Object to perform initialization for those instance Variables.

Interface ⇒ every Variable is always Static, public and final their is not change of existing instance Variable inside the class.

So. we should perform initialization at time of declaration.

So. Constructor is not required for interface.

eg:-
```
Class Person {
    String name;
    Int age;
    Port person ( String name, Int age) {
    this.name = name;
    this. age = age;
    }.
}
```

```
Class Student extends person {
  int roll no;
Student (String name, int age, int roll no) {
   Super (name, age, roll no);
   this roll no = roll no;
}
}
```

Can reference be created in abstract class

```
person p= new Student ("Vinay", 22, 61);
```

Can reference be created for interface?

```
ISame Sample =null;
```

Note:- Every method present inside the interface is abstract, but in abstract class also we take only abstract method then.

what is the need of interface?

```
abstract Class Sample {
    public abstract Void M1();
      "      "      "  M2();
}
interface Isample {
   Void M1();
   Void M2();
}
```

we can replace interface concept with abstract class, but it is not good practice.

eg1 :-

```
interface x{
    ||||
    ||||
}
Class Test implement x {
    ||||
}
Test t = new Test ();
```

i) performance is high

ii) while implementating x we can extends. one more class, through we can bring reusability.

eg2 :-

```
abstract x{
    ||
}
Class Test extends x{
    ||
}
Test t = new Test();
```

performance is low.

while extending X we can't extend anyother class so we can't reuse

Note :- If extending is abstract we can't go for interface.

## Wrapper class :- imp

### purpose

1. To wrap primitives into object form so that we can handle primitives also just like objects.
2. To define several utility functions which are required for the primitives.

**Constructs :-** Almost all the wrapper class have 2 Constructors.

a. One taking primitive type.
b. one taking String type.

eg :- Integer i = new Integer (10);
Integer i = new Integer ("10");

Double $d_1$ = new double (10.5);
Double $d_2$ = new Double ("10.5");

**Note :-** if string argument is not properly defined then it would Result in runtime expception called " NumberFormatException ".

eg :- Integer i = new integer ("ten"); //RE: Number format exception.

Wrapper class and its associated Constructr.

Byte ⇒ byte and string
Short ⇒ short and string
Integer ⇒ int and String
Long ⇒ long and String.

** Float ⇒ float, string and double

Double → double and string.

** Character → Character.

** Boolean ⇒ boolean and string

eg:

```
Float f = new Float (10.5f);
Float f = new Float ("10.5f");
Float f = new Float (10.5);
Float f = new Float ("10.5");
```

eg:

```
Character c = new Character ('a');
Character c = new Character ("a"); //invalid.
Boolean b = new Boolean (true);
//  b1 = new Boolean (false);
//  b2 = new Boolean (True); //CE
//  b3 = new Boolean (false); //CE
//  b4 = new Boolean (TRUE); //EE
```

eg:

```
Class Test {
main () {
Boolean B1 = new Boolean ("yes"); //false
Boolean B2 = new Boolean ("no"); //false
    S.O.P (b1);
    S.O.P (b2);
```

```
S.O.p (b1.equals(b2)); //false .equals(false)->true
S.O.p (b1==b2); false
}
}
```

In case of Boolean Constructor boolean Value be treated as true w.r.t to case Sensitive. part of "true" for all other it would be treated as false

Note:- if we are passing string arguments then case is not important and content is not important.

if the content is case insensitive string of true then it is treated as true inall other case it is treated as false.

Note:- In case of wrapper Class ; toString(), is overriden to print the data.

In case of wrapper class, equals() is overriden. to check the content.

Just like string class ; wrapper classes are also treated as "immutable".

## Immutable Class

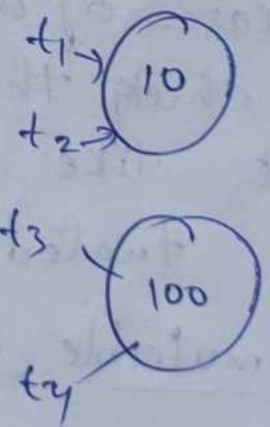
if we create an object and if we try to make a change, with that change new object will be created and those changes will not reflected in old copy.

Can we make our Userdefined class as Immutable?

Ans) Yes possible as shown below.

Ex:

```
final class Test {
    int i;
    Test (int i) {
        this.i = i;
    }
    public Test modify (int i) {
        if (this.i == i)
            return this;
        else
            return new Test(i);
    }
    public static void main (String[] args)
    {
        Test t1 = new Test (10);
        Test t2 = t1.modify(10);
        Test t3 = t1.modify (100);
        Test t4 = t3.modify (100);
        S.o.pln (t1 == t2); // true
        S.o.pln (t1 == t3); // false
        S.o.pln (t2 == t3); // false
        S.o.pln (t3 == t4); // true
    }
}
```



22-11-22

# Wrapper class (conclude)

Wrappers 1.5 v introduced.

* To store primitive data also in object so that we can make use of utility methods. we need wrapper classes.

* Number (parent) → Byte, short, Integer, Long, Float, Double
  2 2 2 2 3 2

* Object (parent) ⇒ Boolean, Character
  2 1 (Char type)

String ⇒ Case and content is not imp

boolean ⇒

Object ⇒
  toString() ⇒ return the hashcode of the object.
  equals() ⇒ Compares the reference.

(wrapper classes) and (String class) are immutable )
  toString() ⇒ print the content of the object
  equals() ⇒ Compares the data.

Can we create our own immutable class?

Ans) Yes,,

—o—

→ In cmd' what ever he is showing methods are available in almost in every wrapper class. (methods overloaded)

public static - helper methods (or) utility methods

→ Wrapper class utility methods

1. ValueOf() method. (cmd javap java.lang.Integer.)
2. XXXValue() method
3. parseXxx() method
4. toString() method.

→ public static wrapper Value Of (String, int) throws java.lang.NumberFormat Exception;

→ public static wrapper ValueOf (String data) throws java.lang.NumberFormatException;

public static wrapper Value of (int data);

Value Of() method.

To create a wrapper object from primitive type or String we use ValueOf() method.

It is alternative to constructor of wrapper class, not suggestable to use

Every Wrapper class, except character class Contain static Value of () to create a Wrapper Object.

eg:1 Integer i = Integer.ValueOf ("10");

Double d = Double.ValueOf ("10.5");

Boolean b = Boolean.ValueOf ("nitin");

S.O.P(i); // 10

S.O.P(d); // 10.5

S.O.P(b); // false

Cg #2

public static valueOf(String s, int radix)

1 ⇒ binary : 2 (0,1)

1 ⇒ octal : 8 (0-7)

⇒ decimal : 10 (0-9)

⇒ hexadecimal : 16 (0-9, a, b, c, d, e, f)

⇒ base : 36 (0-9, a-z)

```
Integer i1 = Integer.valueOf("100",2);
S.O.p(i1);//4 = 2^2//
```

100 ②
$1^2 1^1 2^0$ base

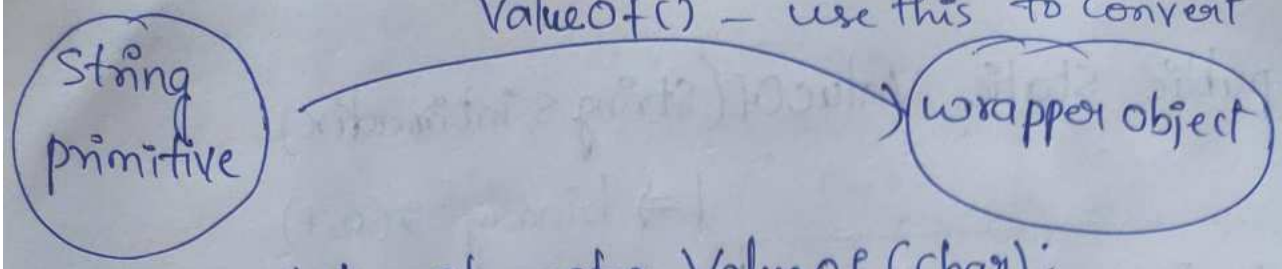
```
Integer i1 = Integer.valueOf("1111");
System.out.println(i1); // 1111

Integer i2 = Integer.valueOf("1111", 2);
System.out.println(i2); //15

Integer i3 = Integer.valueOf("ten");
S.O.p(i3); // RE: NumberFormatException

Integer i4 = Integer.valueOf("1111", 37);
System.out.println(i4); // RE: NumberFormatException.
```

Cg #3 practice

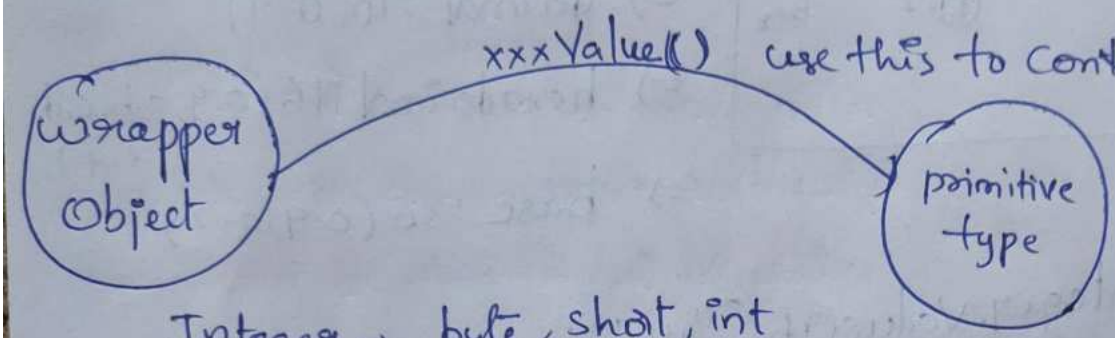
public static valueOf(primitivetype x)

```
Integer i1 = Integer.valueOf(10);
Double d1 = Double.valueOf(10.5);
Character c = Character.valueOf('a');
Boolean b = Boolean.valueOf(true);
```

xx
Primitive (or) String
⇒ valueOf() ⇒
Wrapper object




public static character valueOf(char);




Integer ⇒ byte, short, int
long, float, double → (6 xxxvalue())

↑
Not applicable for Character and boolean

result = minrange + (total - maxrange - 1)
= -128 + (130 - 127 - 1)
= -128 + 2 = -126

(5) xxxValue()

We can use xxxvalue() to get primitive type for the given wrapper Object.

These methods are a part of every Number type object.

(Byte, short, Integer, long, Float, Double) all these classes have these 6 methods which is written as shown below.

## Methods

```
public byte byteValue();
public short shortValue();
public int intValue();
public long longValue();
public float floatValue();
public double doubleValue();
```

eg#1.

```
Integer i = new Integer(130);
// result = minrange + (total - max range - 1)
S.O.p(i.byteValue()); // -126
S.O.p(i.shortValue()); // 130
S.O.p(i.intValue()); // 130
S.O.p(i.longValue()); // 130
S.O.p(i.floatValue()); // 130.0
So.p(i.doubleValue()); // 130.0
```

3. charValue()

Character class contains charValue() to get char primitive for given Character object.

```
public char charValue()
```

eg:1

```
Character c = new Character('c');
char ch = c.charValue();
S.O.p(ch); // c
```

4. boolean Value()

Boolean Class contains booleanValue() to get boolean primitive for the given boolean object

public booleanValue()

eg1

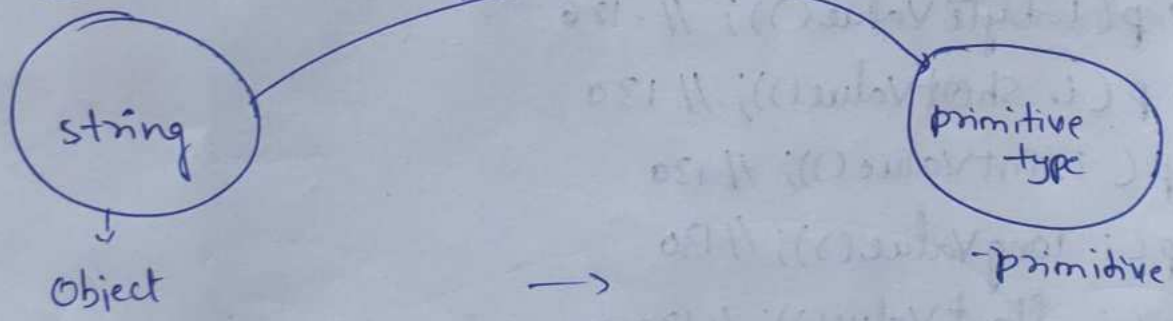
```
Boolean b= new Boolean("nitin");
boolean b1= b.booleanValue();
S.o.p(~~print~~ b1); // ~~false~~
```

* String and Chanact Combinations will not work

In total xxxValue() are 36 in number

⇒ xxxValue ⇒ Convert the wrapper Object ⇒ primitive

(B) parseXXX()




parseXXX()

we use parseInt() to convert String object into primitive type.

form 1.

public static primitive parseXXX(String s)

Every wrapper class, except Character class has parse(xxx) to convert String into primitive type.

eg:- int i = Integer.parseInt("10");

double d = Double.parseDouble("10.5");

Boolean B = Boolean.parseBoolean("true");

~~xxx~~ Usage of Command line arguments. and wrapper class in realtime coding.

// WAP to take inputs from the command line and perform arithematic operations.

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

   // valueOf() ⇒ Converts String / primitive to wrapper type
   // xxxValue() ⇒ Converts Wrapper type to primitive type
   // parseXXX() ⇒ Converts String s to primitive type.

   // Command line arguments ⇒ String inputs = args[0], arg[1]
      int i1 = Integer.parseInt(args[0]);
      int i2 = Integer.parseInt(args[1]);

      S.O.p(i1+i2)
      S.O.p(i1-i2)
      S.O.p(i1*i2)
      S.O.P(i1/i2)
   // args → string, convert into primitive type and process.
}
}
```

form 2

public static primitive parseXXXX (String s, int radix)

|⇒ range is from 2 to 36

Every Integral type Wrapper class (Byte, short, Integer, long) Contains the following parseXXXX() to convert specified radix String to primitive type

eg: int i = Integer.parseInt ("1111", 2);

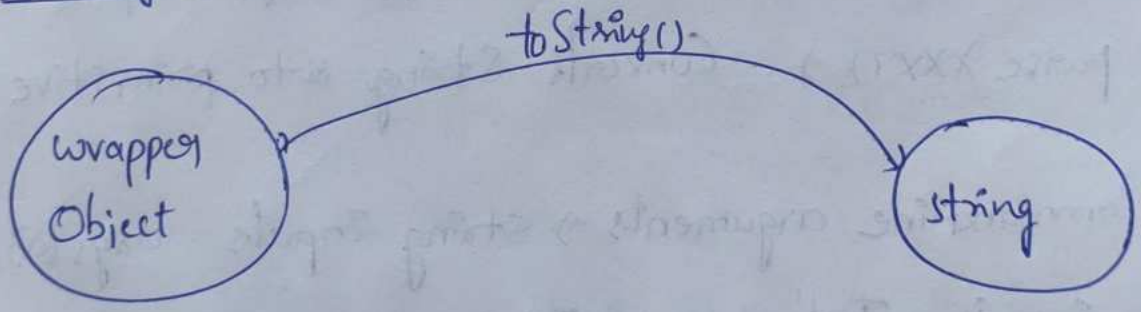
System.out.println(i);//15

Note : ~~String ⇒ parseInt~~xx

String ⇒ parseXXX() ⇒ primitive type.

—o—

→(4) toString()




~~to~~ toString()

TO Convert the wrapper Object (or) primitive to String

Every Wrapper Class Contain ~~s~~ tostring()

form1 :- public String toString()

1. Every Wrapper class (including Character class) contains the above toString() method to convert Wrapper object to string.
2. It is the overriding version of Object class toString() method.
3. Whenever we are trying to print wrapper object reference internally this toString() method only executed.

eg:- Integer i = Integer.valueOf("10");

System.out.println(i); // internally it calls toString() and prints the Data.

form2

public static String toString(primitivetype)

1. every wrapper class Contains a static toString() method to convert primitive to string

String s = Integer.toString(10);
| primitive type int

eg:- String s = Integer.toString(10);
String s = Boolean.toString(true);
String s = Character.toString('a');

(Commonly used
parseXXX
valueof)

form3 Integer and long classes Contains the following Static toString() method to convert the primitive to Specified radix string form

public static String toString (primivitive p, int radix)
                                    | → 2 to 36

eg:- String s = Integer.toString (15,2)
System.out.println (s); // 1111

15
8+4+2+1 = 15
$2^3$ $2^2$ $2^1$ $2^0$
1 1 1 1

form4 :- Integer and Long classes Contains the following to xxxstring() methods

public static string toBinaryString (primivite p);
public static string toOctalString (" ");
public static string toHexstring (" ")

Example

```
class Wrapper class Demo {
    public static Void main (String [] args) {
        String s1 = Integer.toBinary String (7) // 111
        String s2 = Integer.toOctal String (10); // 12
        String s3 = Integer.toHexstring (20); // a
                S.O.P (s1 s2 s3)
    }
}
```

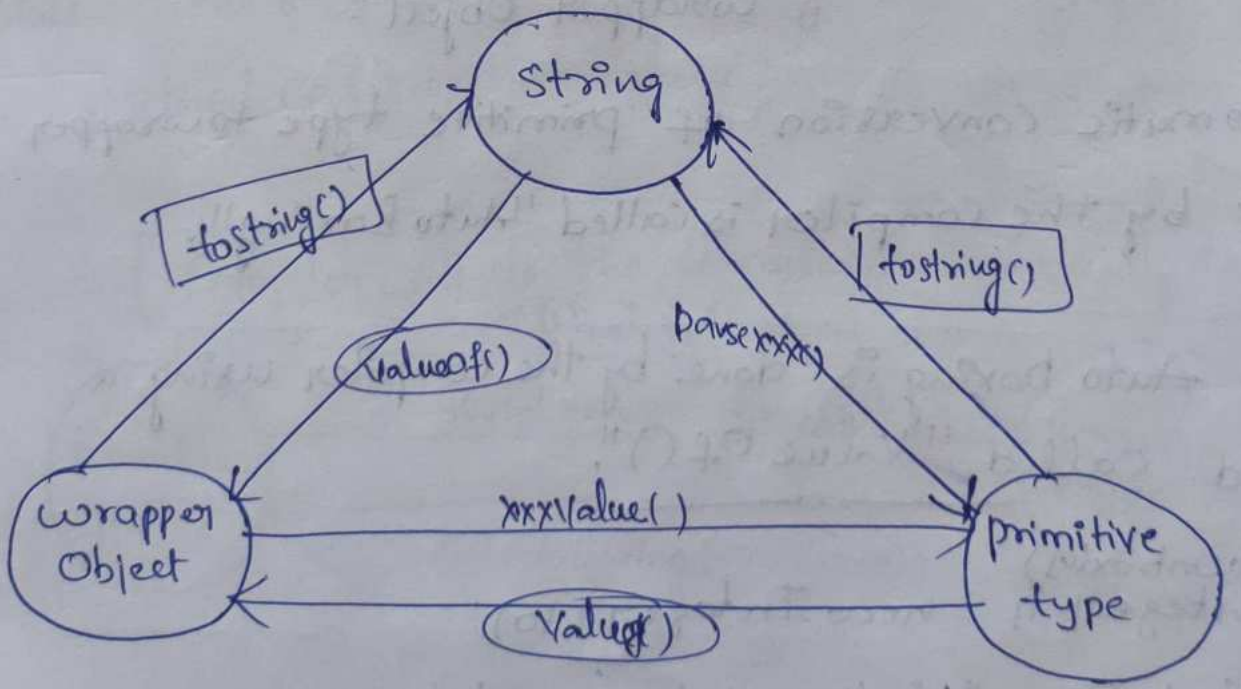
Note :

String class.

```
public static String ValueOf (boolean);
public static String ValueOf (char);
public static String ValueOf (int);
public static String ValueOf (long);
public static String Value of (float);
public static String ValueOf (double);
```

```
String date = String.ValueOf('a'); // Static factory method
String data = "Sachin".toUpperCase(); // Instance factory method
```

(class name → String; object calling → "Sachin")




Conversions

# Auto Boxing and Auto Unboxing

eg#1

```
Boolean b1 = Boolean.ValueOf(true);
if (b1)
    System.out.println("hello");
```

} in jdk 1.4

eg#2

```
ArrayList al = new ArrayList();
al.add(10);
```

Compatability error

Auto Boxing — close it, cover it, put it up in box,

both are same

Auto boxing // actual

```
Integer i1 = 10; // Integer i1 = Integer.valueOf(10);
```

take primivite type wrap it up in object and keep it up in wrapper object.

→ Automatic conversion of primitive type to wrapper object by the Compiler is called "Auto Boxing".

Note: Auto boxing is done by the Compiler using a method called "Value Of()".

(Auto unboxing)

→ xxx `Integer i1 = new Integer(10);`

`int i2 = i1; // whatever data is in wrapped in box we are wrapping it and giving it to primitive`

primitive

↑

(Auto Unboxing - we can call)

Autounboxing

~~int i2 = i1 ; // i1.intValue();~~

(Auto unboxing) Actually in behind scence

int i2 = i1; // int i2 = i1.intValue();

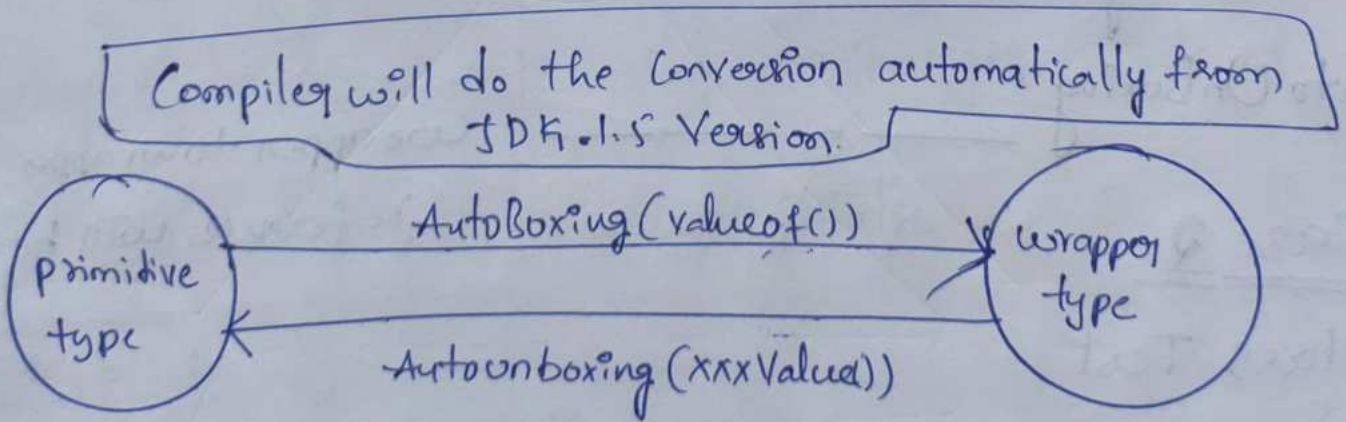
compiler converts Integer to int type using intValue()

## Auto Unboxing

Automatic conversion of wrapper object to primitive type by compiler is called "Auto Unboxing".

```
Integer i1 = new Integer(10);
int i2 = i1;
    |
    | compiler converts Integer to int type using intValue()
    |
int i2 = i1.intValue();
```

Note: Auto Unboxing is done by the compiler using a method called "XxxValue()"



Case 1:-

```
class Test
{
  Static Integer i1=10; // Auto Boxing
  public static void main(String[] args)
  {
    int i2 = i1; //AutoUnBoxing
  }
  public static void m1(Integer i2) // Auto Boxing
  {
    int k= i2; // Auto Unboxing.
    System.out.println(k); //10 - output
  }
}
```

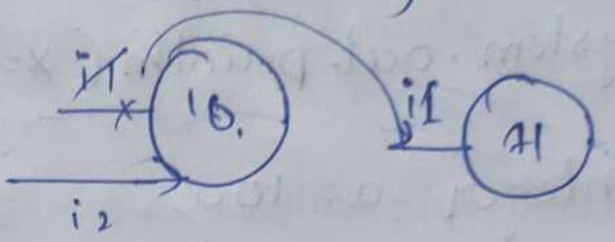
Compiler is responsible for Conversion of primitive to wrapper and wrapper to primitive using the concept of "Auto Boxing" and "Auto UnBoxing" ——— wrapper to wrappy reference will begiven

* Case 2

```
class Test
{
  Static Integer i1; // i1 = null
  public static void main(String[] args)
  {
    int i2 = i1; // int i2 = i1.intValue() :: Nullpointer Excep
  }}  S.O.P (i2);
```

Case 3: All wrapper are immutable

```
Integer i1 = 10; //Auto Boxing
Integer i2 = i1;
i1++; ⇒ i1 = i1+1
System.out.println(i1);
System.out.println(i2);
System.out.println(i1==i2);
```




Case 4 :-

```
Integer x = new Integer(10);
Integer y = new Integer(10);
System.out.println(x==y); // false
```

Case 5 :-

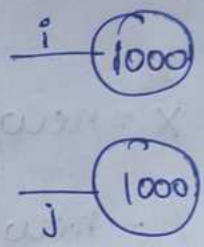
```
Integer x = new Integer(10); //memory from heap area
Integer y = 10; //Auto Boxing ⇒ Integer y = Integer.valueOf(10);
System.out.println(x==y); //false
```

Case 6 :-

```
Integer x = new Integer(10);
Integer y = x; ⇒ Reference is reused so pointing to same object.
System.out.println(x==y); //true.
```

Case 7 :

```
Integer x=10;
Integer y= 10;
System.out.println(x==y);

Integer a= 100;
Iuteger b= 100;
System.out.println(a==b);

Integer i= 1000;
Integer j= 1000;
System.out.println(i==j);
```

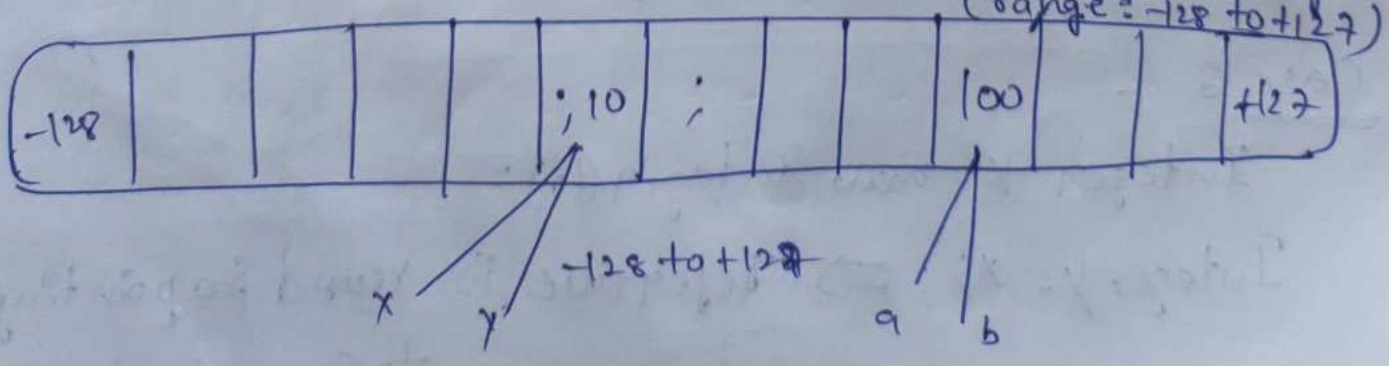



Compiler uses "valueOf()" for Autoboxing

Implemented in intelligent way in wrapper classes

At the time of loading .class file jvm will create buffer of object

Buffer of Objects to be used during AutoBoxing (range: -128 to +127)



Note :- 1. To implement autoboxing concept in wrapper Class a buffer of object will be created at the time of Class loading.

2. During Autoboxing, if an object has to be created first jvm will check whether or not

3. if it is avaiable, the jvm will reuse the buffered object instead of creating a new object

4. if the object is not available inside buffer, then jvm will create a new object in the heap area, this approach improves the performance and memory utilization.

But this buffer concept is applicable only for ~~new~~ few Cases.

1. Byte ⇒ (-128) to (+127)
2. Short ⇒ (-128) to (+127)
3. Integer ⇒ (-128) to (+127)
4. Long ⇒ (-128) to (+127)
5. Character ⇒ (0) to (127)
6. Boolean ⇒ true, false

In the remaining cases new object will be created

// String / primitive to wrapper ⇒ ValueOf()

// wrapper type to primitive ⇒ xxxValue()

23/11/22

## In wrapper class → immutable

equals() ⇒ compares the content

toString() ⇒ print the content

* when compared with constructors it is recommened to use ValueOf() method to create wrapper object

JDK 1.0 - method overloading

JDK 1.4 - ~~no met~~ change in the no. of arguments

new method should be written

JDK 1.5V ⇒ Single method can handle any no. of arguments (all should be of same type)

*→ public void ~~no~~ add (int... x) {

s.o.p ("Var-arg approach");

}

(...) After type before variable

JDK 1.5V called Var-args (ellispe)

## Var arg method

It stands for Variable argument methods in Java language, if we have variable no of arguments, then compulsory new method has to written till jdk 1.4.

But jdk 1.5 version, we can write single method which can handle variable no. of arguments (but all of them same type).

Syntax :- method (datatype ... VariableName)

... ⇒ It stands for ellipse

eg:

```
class Demo                    [ JVM internally uses Array representation
{                               to hold the values of x ]

    public void add (int ...x) {   ( one one array created for every method call )
        System.out.println ("Var-args approach");
    }
}
class Test
{
    public Static void main (String[] args) {

        Demo d = new Demo();
        d.add();
        d.add(10);
        d.add(10,20,);
        d.add (10,20,30);
    }
}
```

d.add(); ⇒  — array Object, no data

output :- Var-arg approach
Var-arg approach
Var-arg approach
Var-arg approach.

(2) Note:- internally the Var arg-method will converted to SingleDimension Arrays, so we can access the Var arg method arguments using index.

(turn page)

~~(next page)~~

eg 2:

```
Class Demo
{
public void methodOne (int ... x) {
    int total = 0;
    for (int i=0; i <x.length ; i++){
        total += x[i]
    }
System . out. println ("The sum is " +total);
}
public Static Void main (String[] args) {
    Demo d = new Demo ();
    d. methodOne ();  // The Sum is 0
    d. methodOne (10);  // The Sume is 10
    d. methodOne (10,20)// The sum is 30.
}
}
```

eg:

```
class Demo
{
public void methodOne (int ... x) {
    int total = 0;
    for (int data :x) {
        total += data;
    }
    system.out.println ("the sum is "+total);
}
```

```
public static void main (String[] args) {
    Demo d = new Demo();
    d.methodOne();  // 0
    d.methodOne (10); // 10
    d.methodOne (10, 20, 30); // 60
}
}
```

Case 1

Valid Signatures

1. public void methodOne (int ... x)
2. public void methodOne (int ...x)
3. public void methodOne (int ... x)

Case 2 we can mix normal argument with var argument

public void methodOne (int, int ... y) (valid)

public void methodone (String s, int ... x)

Case 3 while mixing var arg with normal argument var arg should be always last public void methodOne (int ... x, int ... y); (invalid)

Case 4 In an argument list three should be only one var argument

public void methodOne (int ... x, int ... y); (invalid)

Case 5 we can overload Var arg method, but Var arg method will get a call only if none of matches are found.
(Just like default statement of switch case.)

eg: class Test {
{
public void methodOne (int ... i) {
System.out.println("Var arg method");
}
public Void methodOne (int i) {
System.out.println("Int arg method");
}
public static void main (String[] args)
{
Test t = new Test()
t.methodOne (10); //Int arg method
t.methodOne(); //Var arg method
t.methodOne (10,20,30); //Int arg method
}
}

Case 6 public void methodOne (int ... x) => it is can be replace as int[] x

Case 7:
public void methodOne (int ... x)
public void methodOne(int[] x)

Output: CE because we cannot have two methods with same signature.

# Single Dimension Array Vs Var arg method.

1. wherever singledemension array is present we can replace it with Var arg.

eg :: public static void main(String[] args) ⇒ String...args

2. whenever Var arg is present we connot replace it with single Dimension Array.

eg :: public void methodOne(String... args) ⇒ String[] arg
(invalid)

**Note:-**

m1(int...x)

⇒ we can call to this method by passing group of int values and x will become 1D array (int[]x)

m1(int[]x)

⇒ we can call to this method by passing 1D array only.

Note:

eg1

```
class Test{
    public void methodOne(int...x){
        for(int data: x){
            System.out.println(data);
        }
    }
    public static void main(String... args){
        Test t=new Test();
        t.methodOne(10,20,30);
    }
}
```

(In the above pgm x is treated as one-Darray)

eg:2

```
class Test {
    public void methodOne(int[]... x){
        for(int[] oneD:x){
            for(int element:oneD){
                System.out.println(data);
            }
        }
    }
    public static void main(string... args){
        Test t=new Test();
        int[] a={10,20,30};
        int[] b={30,40};
        t.methodOne(a,b);
    }
}
```

In the above progsam x is treated as 2D array

Note.

methodOne(int... x)

⇒ we can call this method by passing a group of int values / so it becomes 1-D array.

methodOne(int[]... x)

⇒ we can call this method by passing a group of 1D int[], so it is becomes 2-D array.

## Wrapper Class

1. AutoBoxing.
2. Widening (Implicit Typecasting done by the Compiler. (applicable for primitive and wrapper type))
3. Var-Args.

Case 1 :- Widening Vs AutoBoxing

```
Class AutoBoxingAndAutoUnBoxing {
    public static void methodone (long l){
        System.out.println ("widening ");
    }
    public Static void methodOne (Integer i ){
        System.out.println( "autoboxing");
    }
    public static Void main (string [ ] args ) {
        int x=10;
        methodOne (x); // primitive ⇒ do type Casting
                       ⇒ found ⇒
                       long (binding happens by compiler)
    }
}  output : widening.
```

Case 2 widening Vs Var-arg method.

```
class AutoBoxingAndAutoUnBoxingDemo{
    publicStatic void methodone(long l) {
        System.out.println ("widening");
    }
    public StaticVoid methodOne(int... 1){
        System.out.println ("Var-arg method");
    }
    public Static Void main (String[] args) {
        int x=10;
        methodOne (x); // primitive ⇒ do type
                       Casting ⇒
        found ⇒ long(binding happens by compiler);
    }
}
```

Output : widening

Case 3 :- Autoboxing vs Var-arg method :

```
class AutoBoxingAndAutoUnboxingDemo {
    public static void methodone (Integer i) {
        System.out.println ("Auto Boxing
    }
    public static void methodOne (int ... i) {
        System.out.println ("Var-arg method");
    }
    public static void main (String [] args) {
        int x=10;
        methodOne (x); //int ⟹ implicit typecasting ⇒
long, float, double
                    // int ⟹ Autoboxing ⇒ Integer
    }
}
```

output :- AutoBoxing

Case 4 :-

```
class AutoBoxingAndAutoUnBoxingDemo {
    public static void methodone (long l) {
        System.out.println ("long");
    }
    public static void main (String[] args) {
        int x=10;
        methodOne(x); //CE can't find the method.
    }
}
```

Note :- Widening followed by AutoBoxing is not allowed in Java, but Autoboxing followed by widening is allowed.

Case 5:

```
Class AutoBoxingAndAutoUnboxingDemo{
    public static void methodOne (object o) {
        System.out.println ("object");
    }
    public static void main (String[] args) {
        int x=10;
        methodOne(x); // AutoBoxing → int ⇒ Integer
                      // widening ⇒ Integer ⇒
                                    Number, Object
    }
}
```

output: object

which of the following declarations are valid.

1. int i=10; // valid
2. Integer I=10; // Autoboxing (valueof())
3. Int i=10L; // invalid (long ⇒ int)
4. Long l=10L; // Autoboxing (valueOf())
5. long l=10; // Autoboxing ⇒ Integer ⇒ Number, Object so invalid.
6. long l=10; // valid (int ⇒ long)
7. Object o=10; // Autoboxing ⇒ Integer ⇒ Number, Object so valid
8. double d=10; // valid (int ⇒ double)
9. Double d=10; // AutoBoxing ⇒ Integer ⇒ Number, Object, So invalid
10. Number n=10; // Auto Boxing ⇒ Integer ⇒ Number, Object so valid.

# new Vs newInstance()

1. new is an operator to create an object an objects if we know class name at the beginning new operator
2. newInstance() is a method presenting class "Class", which can be used to create object.
3. if we don't know the class name at the begining and its available dynamically Runtime then we should go for newInstance() method.

eg 1 :-

```
public class Test {
    public static void main (String[] args) throws
        exception {
    // Take the input of the Classname for which object
has to be created at the runtime
        String className = args[0];
    // load the class file explicity (whatever I am giving command line)
    Class c = Class.forName(className);
                    (to load class methods)
                    method - in class
    // for the loaded Class object is created using
Zero param Constructor only. (whatever load new instance will be created)
        Object obj = c.newInstance();
    // perform type casting to get Student Object.
        Student std = (Student) obj;
        System.out.println(std);
    }
}
```

if dynamically provide class name is not availabl then we will get the RuntimeException saying ClassNotFoundException

To use newInstance() method compulsory corresponding class should contains no argument constructor. otherwise we will get the RuntimeException saying "InstanceNotFoundException".

if the argument constructor is private then it would result in "IllegelAccessException". (CE-shouting)

Note:- During typecasting, if there is no relationship b/w 2 classes as parent to child then it would result in "classCastException",

Difference b/w new and newInstance();

new

new is an operator, which can be use used to create an Object.

we can use new operator, if we know the class name at the beginning.

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

If the corresponding .class file not available at Runtime then we will get RuntimeException saying NoclassfoundError, it is unchecked.

To used new operator the corresponding class not required to contain no argument constructor.

## newInstance().

newInstance() is a method, present in class class, which can be used to create an object.

we can use the newInstance() method, if we don't class name at the begining and available dynamically Runtime.

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

If the corresponding .class file not available at Runtime then we will get RuntimeException saying ClassNotFoundException, It is checked

To used newInstance() method the corresponding class should Compulsory contain no argument constructor, otherwise we will get RuntimeException saying Instantition.Excepti

## Difference blw ClassNotFoundException & NoclassFounderr

1. for hard cdded class Names at Runtime in the corresponing .class files not available we well get NoclassDefFoundError. which is unchecked.

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

In Runtime Test.class file is not available then we will get NoclassfoundefError.

2. for Dynamically provided Class names at Runtime, if the corresponding .class files is not available then we will get the RuntimeException saying

"ClassNotfoundException".

Ex:- Object o = Class.forname("Test").newInstance();

At Runtime if Test.class file not available then we will get the ClassNotfoundException, which is checked exception.

Note:- new will create a memory on the heap area.

Student ⇒ JVM will search for student.class file in Current working Directory if found load the .class file data into Method Area

During the loading of .class file.

a. Static Variable will get memory set with default value.

b. static block gets executed.

In the heap area. for the required object memory for instance Variables is given jvm will set the default values to it.

a. Execute the instance block if available

b. Call the Constructor to set the meaningful Values to the instance Variables.

JVM will give the address of the object to hashing algorithm which generates the hashcode for the object and that hashcode will be returned as the reference to the programmer.

new ⇒ required class details known to compiler but not available at JVM then it would result in "No Class Def found Error".

NewInstance() ⇒ required class details not available at jvm then it would result in "Class Not found Exception".

25/11/22 :

1. import

| Implicit | Explicit |
| --- | --- |
| java.util.*; <br> Arraylist L | java.util.*; <br> ArrayList <br> ArrayList; |
| Complitime high | Complitime low |

* javac ⇒ search for the required class information in a) current working directory

b) Did the programmer specified where the class is available.

→ we explicitly have import the package in where class is present.

java.util.ArrayList al = new java.util.ArrayList();

fully quilafied path.

import means - bring the class to our current working directory

informing the Compiler plz search for ArrayList Class in "java.util"

import java.util.ArrayList; // Explicit import
import java.util.*; // implicit import

---

## Lambda Expression : interface functionality

###

## Import Statement

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

Output:
compile time error
because ArrayList (class) is not there in current directory.

- we can resolve this problem by using fully qualified name "java.util.ArrayList"
  l = new java.util.ArrayList(); " . But problem with using fully qualified name every time is it neccesary length of the code and reduces readability.

⇒ we can resolve this problem by using import statements

Example :-

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

Output: D:\Java> Javac Test.java.

Hence whenever we are using import statement it is not require to use fully qualified names we can use short names directly.
this approach decreases length of the code and improves readability.

Case 1: Types of Import Statements:
There are 2 types of import statements.

1) Explicit class import
2) IMplicit class import

Explicit class import:

Example : Import Java.util.ArrayList;

⇒ This type of import is highly recommended to use because it improves readability of the code

⇒ Best suitable for developers where readability is imp.

Implicit class import:

Example: import Java.util.*;

⇒ It is never recommened to use because it reduces readability of the code

⇒ Best suitable for students where typing is important

Case 2:

Which of the following import statements are meaningful

a. import java.util;
b. import java.util.ArrayList.*; (c and) d
c. import java.util.*;
d. import java.util.ArrayList;

Case 3: Consider the following code.

```
Class MyArrayList extends java.util.ArrayList
{
}
```

⇒ The code compiles fine even though we are not using import statements because we used fully qualified name.

⇒ whenever we are using fully qualified name it is not required to use import Statement.

Similarly whenever we are using import statements, it is not require to use fully qualified name.

Case 4 :

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

The output : CE

reference to Date is ambiguous both class java.sql.Date in java.sql and class java.util.Date in java.util match.

Date d= new Date();

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

Case 5 : while resolving class names Compiler will always gives the importance in the following order.

1. Explicit class import
2. Classes present in current working directory
3. Implicit class import.

Example :-

```
import java.util.Date;
import java.sql.*;
```

```
Class Test {
    public static void main (String args[]){
        Date d= new Date();
    }
}
```

The code compiles fine and in this case util package Date will be considered.

Case 6 :-

whenever we are importanting a package all classes and interfaces present in that package. are by default available but not sub package. classes.

Java
|⇒ util
|⇒ Scanner.class, ArrayList.class, LinkedList.class.
|⇒ regex
|⇒ pattern.class.

To use pattern class in our program directly which import Statement is required?

a. import java.*;
b. import java.util.*;
c. import java.util.regex.*; //valid (implicit)
d. import java.util.regex.pattern; //valid (explicit)

Note :

* ⇒ it refers to only .class files not sub packages .class files.

Case 7 : In any java program the following 2 packages are not require to import because these are available by default to every java program.

1. Java.lang package
2. default package (current working directory)

Case 8 :- "Import statement is totally Compile time concept" if more no of imports are there then more will be the compile time but there is "no change in execution time".

Difference b/w C language #include and java language import?

1. It can be used in C & C++
2. At Compile time only compiler copy the code from Standard library and placed in Current program
3. It is Static inclusion.
4. wastage of memory.

Ex: <jsp:@ file = "111" >

## import

1. It can be used in Java.
2. At runtime JVM will execute the corresponding standard library and use it's result in current program.
3. It is dynamic inclusion
4. No wastage of memory

Ex: <jsp:include>

Note:- In the case of C language #include all the header files will be loaded at the time of include statement hence it following static loading

But in java import statement no ".class" will be loaded at the time of import statements in the next lines of the code whenever we are using a particular class then only corresponding ".class" file will be loaded Hence it follows "dynamic loading" or "load-on-demand" (or) "load-on-fly".

JDK 1.5 Versions new features.

1. for each
2. Var-arg
3. Queue
4. Generics
5. Autoboxing and Auto Unboxing.
6. Co-variant return types.

7. Annotations
8. Enum
9. static import
10 String builder.

Static import: This concept introduced in 1.5v

According to sun static import improves readability of the code but according to worldwide programming exports (like us) static imports creates confusion and reduces readability of the code. Hence if there is no specific requirement never recommended to use a static import.

Usually we can access static members by using class name but whenever we are using static import it is not require to use class name we can access directly.

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());
    }
}
```

Output :- 2.0 , 20, 0.8413061...

* 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 ());
    } }
```

output:- 2.0, 20, 0.4302285.

```
Class Test {
    static String name = "sachin";
}
```

output :- Test.name.length() ⟹ 6,,

***

```
import java.io.PrintStream;
Class System {
        static PrintStream out;
}
Class PrintStream {
        public void println (){
            ;;;;;
    } }
```

System.out.println()

- out → it is a reference of printstream class.
- println() → it is a method of printstream class
- System → it is a class

Example 3:

```
import static java.lang.System.out;
Class Test {
    public static void main (String args[]){
        out.println ("hello");
        out.println ("hi");
    }
}
```

output : hello hi

Example 4:

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
Class Test {
    public static void main (String args[]args){
        System.out.println (MAX_VALUE);
    }
}
```

output : Ambiguous ⇒

Byte match (MAX_VALUE);

Note:

Two packages Contain a class or interface with the same is very rare hence ambiguity problem is very rare in normal import.

But 2 classes or interfaces Can Contain a method (or) Variable with the same name is very common hence ambiguity problem is also very common in static import while resolving static members compiler will give the precedence in the following order.

1. Current class Static member
2. Explicit static import
3. implicit static import.

eg: import static java.lang.Integer.MAX_VALUE;
import static java.lang.Byte.*;

```
class Test {
    static int MAX_VALUE = 999;
    public static void main(String[] args) {
        System.out.println(MAX_VALUE);
    }
}
```

Which of the following import statement is valid?

import java.lang.math.*; //invalid.
import static java.lang.math.*; //valid.
import java.lang.math; //valid.
import static java.lang.math; //invalid.
import static java.lang.math.sqrt.*; //invalid.
import java.lang.math.sqrt; // invalid.
import static java.lang.math.sqrt(); //invalid.
import static java.lang.math.sqrt; @ //valid.

Usage of static import reduces readability and creates confusion hence if there is no specific requirement never recommended to use static import

what is the difference b/w general import and static import.

normal → we can use normal imports to import classes and interfaces of a package

⇒ whenever we are using normal import we can access class and interfaces directly by their short name it is not require to use fully qualified names.

static import

→ we can use static import to import static members of a particular class.

⇒ whenever we are using static import it is not

not require to use class name we can access

Static members directly.

## Lambda Expression

lambda expression → functional interface → interface
inheritance Abstraction.

In interface if one abstract method is present we can call it has funtional interface

```
interface Demo {
    Void disp();  → functional interface
}
```

functional interface and lambda expression work together.

@ functional Interface → It will inform to Compiler and developer like it is an functional interface

```
@ functional Interface
interface Demo {
    Void disp();
}
```

```
@ functional Interface
interface Demo {
    Void disp()
    Void disp()
}
```

Compiler error beco2 we mentioned above as functional interface. It means that interface hay only method.

we can't lambda expression without function Interface.

Lambda Expression :- a short block of code which takes in parameter and returns a value.

In Java 8

→ ⇒ arrow operator
↳ lambda Operator

Before

```
void disp(){
S.o.p("Hello");
}
```

To write lambda expression
we need to use (→)
- minus and > greater.

Anonymous method means unname method.

```
()  →  { }
|        └ Body
parameters
lambda Operator
```

we need to write lambda expression
i) use →
ii) divide 2 parts.
a. left of lambda operator parameters required.
b. Right of lambda operator body required {}

```
() → { S.o.p("Hello"); };
```

normal method

```
Void disp()
    S.O.p ("Hello");
}
```

in method we have only one statement then array {} brackets we can ignore

```
() → S.O.p ("Hello")
```

lambda Expression can't be write alone lambda expression and functional interface work together.

Ex:

```
@functionalInterface
interface Demo()
  Void disp();
}
main(){
Demo d = () → S.O.p ("Hello"); (without {} because one statement is present)
   d.disp();
(or)
Demo d = () → {
   S.O.p ("Hello");
  };
d.disp();
```

(we can write with {} even one statement is there. There is no issue);

For functional interface we can write anonymous ours inner class instead of lambda expression

Annoymous inner class

```
Clas Demo{
    void disp();
}
main (){
    Demo d = new Demo(){
        public void disp(){
            S.O.p ("Hello");
        }
    };
    d. disp();
```

For functional interface total 3 ways are there 1. Normal Tradition

2. Lambda Expression
3. Annoynomous inner class

But lambda expression is able to write only one abstract method is present.

Parameter :-

```
@ Functional_Interface
interface Demo{
void add (int a, int b);
}
```

```
main(){
  Demo d = (int a, int b)
        int res = a+b;
        s.o.p (res);
    };
  d. add (10,20);
```

→ we can remove

Parameter & Return type.

```
interface Sub
  {
    int sub (int num1);
  }
  main() {
     Sub s= (num1) → {
        int res = num1-5;
        return res;
    };
    s. sub (10); → we can store also
  s.o.p (s. sub (10)); → we can directly print value.
```

if one parameter is there then no need of ()

num1 → num1-5;

return statement is not needed.

i) To write lambda exp we need to use lambda operator →

ii) lambda operator divided into parts to write lambda exp.

iii) left side of lambda operator () => are needed.

4) Right side of lambda operator Body => is need

5) left side if single parameter is there no need of ()

6) left side for parameters data types is optional.

7) Right side of body contains one statement than {} is optional

8) Right side in body if single line implementation then return type is optional.

9) {} is mandatory in case if there is return statement explicity present and in Body more than one statement is present

// W.A.P to compute length of string.

3) ways to:

```
Interface Demo{
    int add (int a, int b);
}
Class Demo1 implements Demo{
    public int add (int a, int b) {
        int res = a+b;
        return res;
    }
}
```

main()

Demo d= new Demo1();

d.add (10,20); To store return value

S.O.P (d.add (10,20)); to display return value.

Lambda

```
main() {
    Demo d= (a,b) → res =a+b;
    S.O.P (d. add (10, 6));
```

Inner class

```
main() {
    Demo d= new Demo () {
        public int add (int a, int b){
        int res =a+b;
            return res;
    } }
} }
S.O.P ( d.add (10,20));
```

———— o ————

25/11/22.

# Method hiding :-

→ Static methods do participate in inheritance yes, but we can't override if we do that it will consider as specialized method.

→ & it is called method hiding.

→
```
class parent {
    public void static disp()
    public static void disp(){
        S.O.P ("Hello parent");
    }
Class Child extends parent
{
    public static void disp()
    {
        System.out.println ("hello child1");
    }
}
public class methodHiding {
    public static void main (String [] args) {
        parent p= new Child1();
        p.disp();
    }
}
```

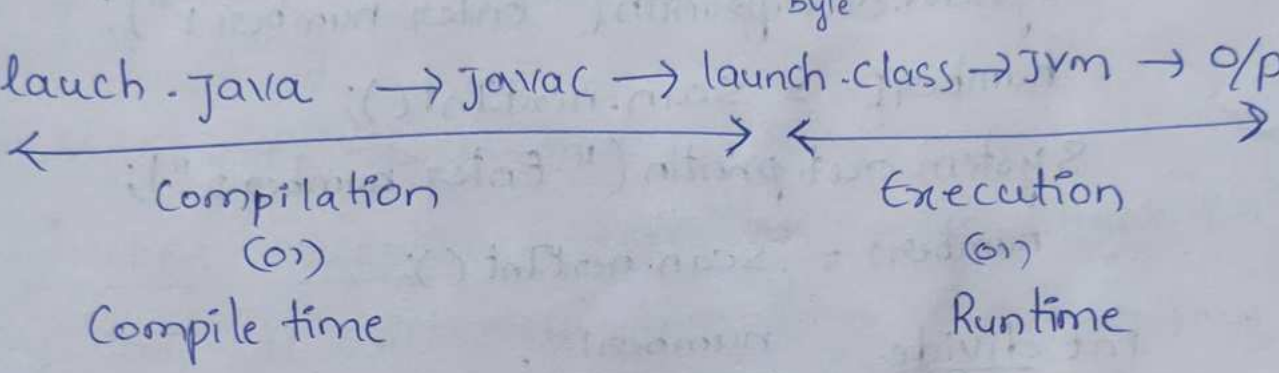
output : Hello parent.. (we can't override Static methods)

## *** Exceptional Handling :-

Java program if it is executing that means it is running.

→ try, catch, throws, throw, finally

web App :- without internet we can't run the app




error :- Because of Some language Syntax (;, { }, void) it will check at Compilation compile time error

Runtime error :- i) Runtime error → because of logical mistake ex: a/b = 10% (Arithmetic error at runtime)
↓
logical mistake (by zero it is not)

Exception :- ① Default input by user (or) developer In between only application will exit is called exception.

② Exception refers to mistake that will occur during runtime of our application. which will result in Abrupt termination.

## exception trample① (type and check this code)

Ex:
```
import java.util.*;
public class Demo {
    public static void main (Strings [ ] args){
        Scanner scan =new Scanner (System.in)
        System.out.println (" divided operation'app");
        System.out.println ( "Enter number 1 ");
        number1 = scan.nextInt( );
        System.out.println (" Enter number 2");
        number2 = scan.nextInt ( );
    int divide = number1 / number2 ;
        System.out.println (divide);
    } System.out.println ("operation over"));
}
```

(divide: variable to result)

output; user entering numbers

① case①
number1 = 100
number2 = 10
output = 10

② case
number = 100
number = 0
output = —

Exception will come means abruptly terminated by telling (Arthmetic exception by zero / zero. at main method) like this.

So we have handle exception without Abrupt termination, then exception handling came to picture

Exception Handling:- It is process to handle not to Abrupt termination should not happen.

try - catch - throws - throw - finally ⇒ keywords we are going to use in Exception handling.

Exceptions are two types

1) Checked exception :- Compiler only check for risky code it will tell like (hey developer your having some risky please handle it).

2) unchecked exception:- Compiler will not check it will tell like (hey devoloper your ~~can~~ headache I am compiling and giving you. you do with JVM at Runtime.) During Runtime it will check and throw an exception. like example ①.

→ How to handle expection ?- refer example 2

Ans:- try {

}

∴ Both are we should write
we can't write only one alone

Catch (inbuilt classes ~~var~~ variable ) we should give means (reference)

{

}

it will handle try block

try { } - in try block what ever risk code we have to write there

Catch() { } - exception handler.

* Exception handling in this example

Example 2

```
public class {
    public void static main (String [] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println ("division operation app");
        try {                                   (thread will execute line by line)
            System.out.println ("Enter num1");
            num1 = scan.nextInt()
            System.out.println ("Enter num2");
            num2 = scan.nextInt();
            int res = num1/num2;
            System.out.println(res);
        }
        Catch (exception e) {
            System.out.println(" please don't enter non-zero deno");
        }
        System.out.println(" app operation terminated");
    }
}
```

keeping in try block

risky code

output: Case① num1 = 100, num2 = 10 → res = 10; operation over

Case②

num1 = 100

num2 = 0

output: Please don't enter non-zero denominator

operation terminated.

try{} :- if any risky code is there then we will put in try block.

catch(){} :- it will handle the try block

---

if there is any exception then only catch block will execute otherwise try block will execute.

---

if exception statement is there in try block then from that exception line till catch block block line no statement will not excute. directly Catch block excute.

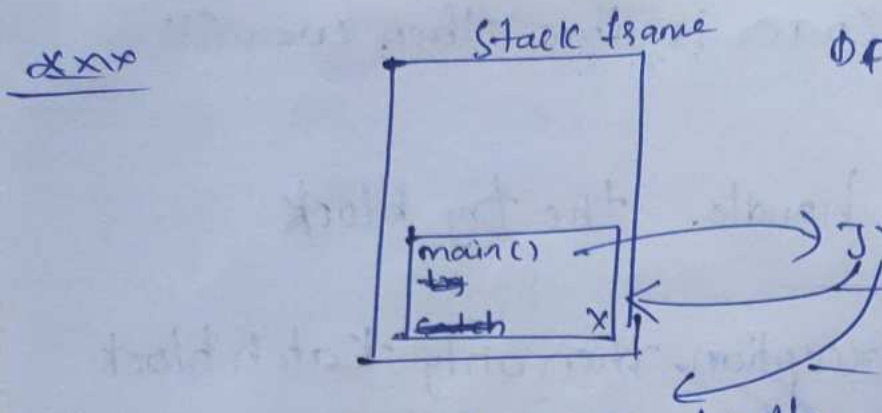
Ex :- 10 lines code

```
try{
  (5) line → exception → then 6, 7, 8 will not excute
6                        directly catch block (9)
7                        and after catch
8                        block lines
catch() (9) line → catch block
{
}
```

(10) - After catch block

If there is no exception in try the catch block will not execute

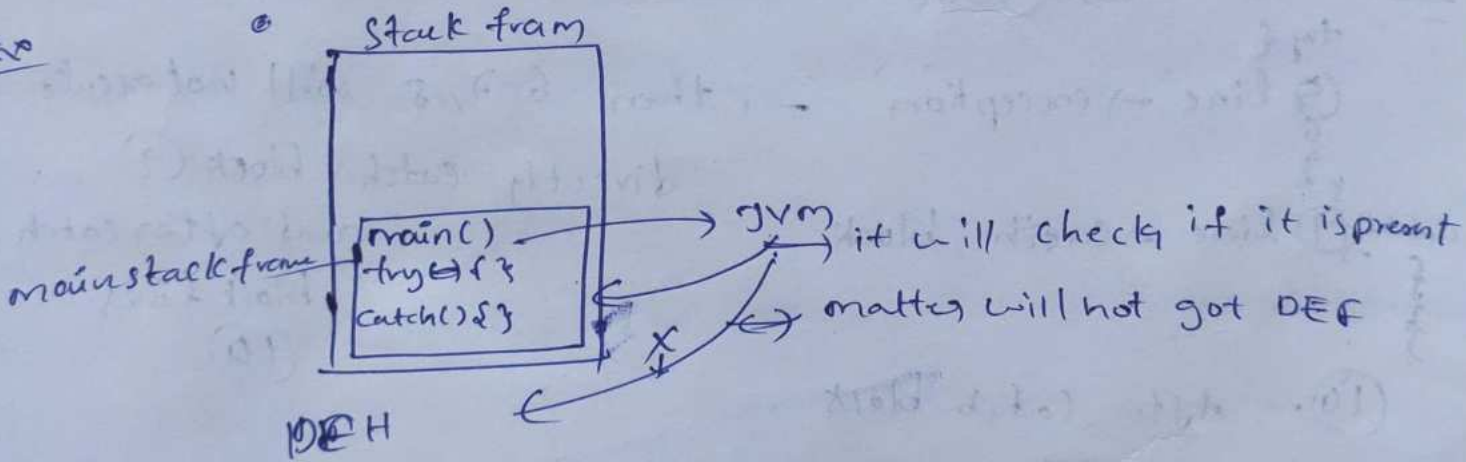
if exception or no exception but after catch block lines will execute.




DEH → Default exception handler.

it will check if not matter come to DEH then DEH throw exception

when ever some exception raises in ~~main method~~ stack frame where ever that exception line it will ~~throw~~ create object and throw to JVM, then after JVM will check again in that stack frame like wheather progsammer handled exception or not, if not itw go to default exception handler.




when ever some exception raises in ~~main method~~ stack frame where ever that exception line it will create object and throw to JVM, then after JVM will check again in that stack frame like wheather progsammer handled or not, if he handled it will not go default exception handler.

For one try block we can write no. of catch block.

For one try block we can write only one catch block also.

— o —

## day 2 "Exception handling"

***

### Example① with cases

```
statement -1;
try {
   Statement -2;
   Statement -3;
   Statement -4;
}
catch(xxx e)
{
  statement-5;
}
statement-6;
```

Case 1

* "if there is no exception raised"
→ statement-1,2,3,4 & 6 will be executed Resulting in normal Termination

* "if an Exception is raised at statement-3 and the corresponding catch block is matched."
→ statement-1,2,5 & 6 will be executed Resulting in Normal Termination.

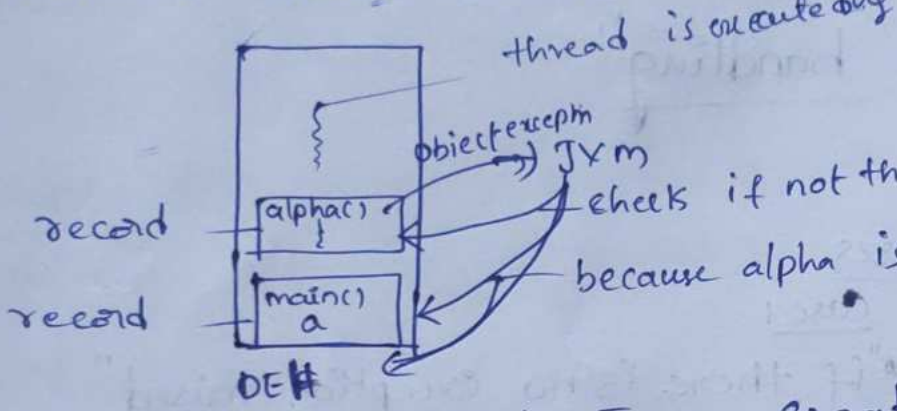
* "If an Exception is raised at statement-4 and the corresponding catch block is not matched."
→ Statement-1,2 will be executed. Resulting in Abnormal Termination

* "If an Exception is raised at statement-1 or statement-5 or statement-6."
→ Statement-1 or 5 or 6 is not a part of try block Resulting in Abnormal Termination.

— o —

→ We create class in that one Alpha method with some code,
and from main method we are calling the Alpha method.

Runtime Stack is there




In main method Iam creating object and calling the Alpha object.
if anything wants execute the entire code bought to runtime stack the excute will excute.
if alpha method contains exception it will create alpha object give it JVM,
and jvm will again in alpha method wheather is exception handled or not. if not handled the jvm will check in main method because main method is calling alpha.

xxx
Note :- In Alpha it will check if ~~not~~ exception not handled in Alpha, JVM will check in main method, main method is one who is calling Alpha, if it is ~~not~~ exception not handled in main it goes to default Exception handling.

Example exception

Exception in thread "main" java.lang.AsthmeticException:/
by zero at Alpha.alpha(lauchException4.java:15)
at LanchException4.main(LaunchException4.java:26).

(DEH)
hey in main thered Some exception is there
matter went to Alpha and After main method they both not handdled so matter came to Default exception handler.

---

⇒ whenever there is an Exception

① Handle Exception (try{}-catch(—>){}

② Duck the Exception (throws)

③ Re-throwing an Exception (throw, throws, try, catch(), finally)

⇒ {try, catch, throw, throws, finally}

throws expection

Ex:-
```
class Vijay{
  Class Alpha()
  {
  int a=10
  int b=0
  int c=a/b; - Expection
  So.p(c);
  }
}
```

```
Beta
class ~~Vijay~~ {
  Void beta() throws excepetion
  { Alpha a=new Alpha();
  } a.alpha();
}
public class Vijmain{
  public static Voidmain(){
    Beta b= new Beta();
      b.beta
  }
}
```

In above example

we have to write throws exception if your not handling the exception for good practice. because who is the caller of that method no need check entire code.

when it is calling only, it get to know exception is there it has to handle and again it is not handled then who is calling it will go to them.

if no one is handling it will go to default exception handling (DEH)

---

It will throws exception automatically but to be mature write (throws exception);

who ever writing code he has to handle the exception compulsory / no ducking)

Ducking means throwing exception without handling to who calling that. If it is unchecked exception

---

Unchecked :- Compiler will not force us to handle

Checked :- Compiler will force to handle that exception

ex:- Thread.sleep(4000); ——— compiler error
it will sleep for 4sec
nothing will happen but still Compiler will force to handle. ——— just indication for caller
Now duck it (throws exception)

throw → Re throwing → throwing already handled exception. object to Caller
ref variable
- in the th catch block write (throw e;)

After throw keyword whatever it is it will not execute.
So execute @ that we have to write in finally block &(finally { }) — to close
finally block we can't write alone with try block we can.

* try & catch → to handle exception
throws → duck @ method signature
throw → Catch → rethrow
finally → Close resources

→ What is exception object Contain

Name of the exception: Arthmetic exception

Description of the exception: / by zero

Stack Trace of Demo.alpha (Lauch.java:22)
the exception: lauch.main (lauch.java:7)

Methods to print exception Information.

Throwable
getMessage()
toString()
printStackTrace()

1. e.getMessage() prints the description of the exception
Example: / by zero

2. e.toString() prints the Come name and the description of the Exception

Example: ArithmeticException: / by zero

3. e.printStackTrace() prints the name and the description of Exception along with the stack trace
Example: ArithmeticException: / by zero
at Demo.alpha().

finally → ① if no exception (✓) - encute

→ ② if exception & catch match (✓)

→ ③ if exception & catch not match (✓)

to close connection we use finally { }

After return statement below statement is will not execute.

So we write in finally block it will dominate the return and execute.

System.exit(0); — total JVM will close

total Java Application will close.

| Exception Raised | Exception Handled | finally block executed |
|---|---|---|
| NO ✗ | — | Yes ✓ |
| yes ✓ | Yes ✓ | Yes ✓ |
| Yes ✓ | NO ✗ | Yes ✓ |

finally block executing types

Ex

```
Statement-1;
try
{
  Statement-2;
  Statement-3;
  Statement-4;
}
Catch (xxxe)
{
   Statement-5;
}
finally
{
  Statement-6;
}
Statement-7;
```

DEH at last

Cases

Case 1 → if there is no Exception raised
→ Statement-1,2,3,4,6 & 7 will be executed
Resulting in Normal Termination.

Case 2 if an exception is raised at Statement-3 and the corresponding catch block is matched.
→ Statement-1,2,5,6,7 will be executed
Resulting in Normal Termination.

Case 3 if an exception is raised at ~~Stat~~ Statement 3 and catch block matched.
→ Statement-1,2,5 & 6 will be executed
Resulting in Abnormal Termination

Case 4 if an Exception is raised at Statement-5
→ Resulting in Abnormal Termination
But before that finally block will be executed.

---

Differences b/w throw & throws.

| throw | throws |
|---|---|
| → throw keyword is used to explicitly throw an exception to the JVM. | → throws keyword is used to declare an exception to delagate exception handling responsibility to the caller |
| → throw keyword is followed by an instance of throwable or a subclass of Throwable. | |

→ throw keyword is used within the method body

→ multiple exceptions can't be thrown with a single thrown clause

→ used in rethrowing an exception.

```
try {
}
catch (Exception e) {
    throw e;
}
```

throws keyword is followed by exception class name

throws keyword is used with the method signature

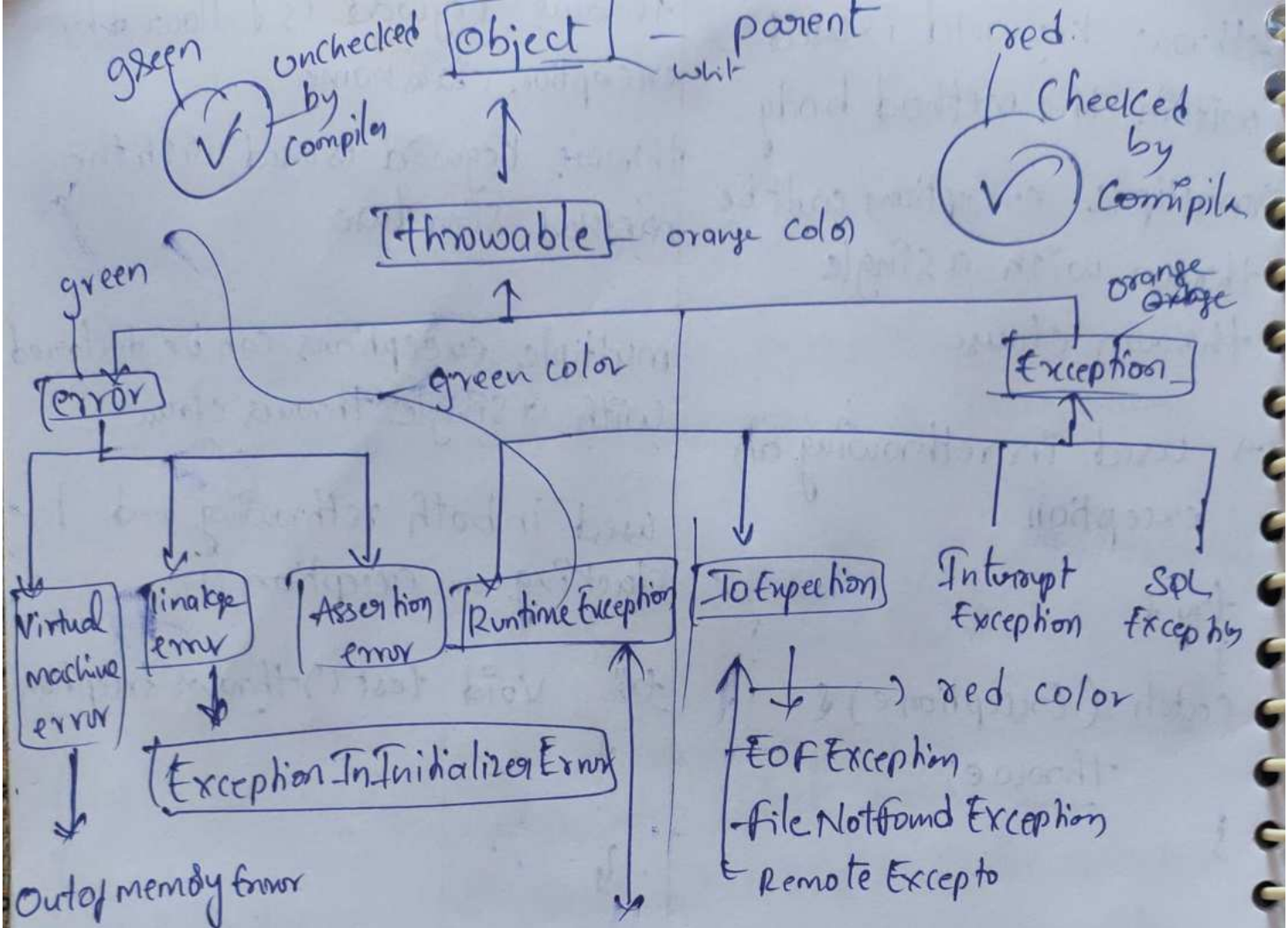
multiple exceptions can be declared with a single throws clause.

used in both rethrowing and ducking an exception

```
try  void test() throws Exception
  {

  }
```

## Exception Handling

```
try
{
    Risky / Suspicious code
}
catch ( Exception e)
{
    Handling / Alternate / pre cautionary code
}
finally
{
    Resource deallocation / clean up code.
}
```




green → Unchecked exception

red → fully checked Exception

orange → partially checked mean both

So. Exception Class is partially checked.

# day 3 exception handling

## Nested try - catch and finally

```
try {
   Statement-1;
   Statement-2;
   Statement-3;
   try {
      Statement-4;
      Statement-5;
      Statement-6;
   }
   catch (xxx e) {
      Statement-7;
   }
   finally {
      Statement-8;
   }
   Statement-9;
}
catch (yyy e) {
      Statement-10;
}
finally {
   Statement-11;
}
Statement-12;
```

**case 1** : if no exception occurs

→ Statements-1,2,3,4,5,6,8,9,11 & 12 will be executed Resulting in Normal Termination

**Case 2** if an exception occurs at Statement-2 and the corresponding catch block is matched.

→ Statements-1,10,11 & 12 will be executed. Resulting in Normal Termination

**Case 3** : if an exception occurs at Statement-2 and the corresponding catch block is not matched.

→ Statements-1 & 11 will be executed Resulting in Abnormal Termination.

**Case 4** : If an exception occurs at Statement-5 and the corresponding inner catch block is matched.

→ Statements-1,2,3,4,7,8,9,11 & 12 will be executed. Resulting in Normal Termination

**Case 5** : If an Exception occurs at statements and the corresponding inner catch block is not matched, but outer catch block ma

→ Statements-1,2,3,4,8,10,11,& 12 will be executed, Resulting in Normal Termination.

Case 6 :- If an Exception occurs at statements-5 and both inner and outer catch blocks are not matched.

→ Statements -1, 2, 3, 4, 8 & 11 will be executed Resulting in Abnormal Termination.

Case 7 :- If an Exception occurs at statement-7 and the corresponding catch block is matched.

→ Statements- 1, 2, 3, (Exception may occur at one of the Statements - 4, 5 or 6) 8, 10, 11, 12 will be Executed. with normal termination.

Case 8 :- If an Exception occurs at statement-7 and the Corresponding catch block is not matched.

→ Statements - 1, 2, 3, E Exception may occur at one of the Statements - 4 or 5 or 6), 8, 11 will be executed with Abnormal termination.

## Various possible Combinations of try-catch-finally

Case 1

only try

```
try
{
}
```

(not Valid)

---

Case 2

only catch

```
catch(xxx e)
{
}
```

(not valid)

not valid and Valid Syntax of try-catch-finally

Case 3

only finally

```
finally
{
}
```

(not valid)

Case 4 try-catch

```
try
{
}
catch(xxx e)
{
}
```

(valid)

Case 5 :

Reverse order

```
Catch (xxx e)
{
}
try
{
}
```

(Not valid)

Case 6 : multiple try

```
try { }
try { }
catch (xxx e){ }
```

(Not valid)

Case: 7

multiple try

```
try {}
catch (xxx e)
  {
}
try
{
}
```

(Not valid)

Case: 8

multiple try-catch

```
try { }
catch (xxx e) { }
try { }
catch (xxx e) { }
```

(Valid)

Case 9

multiple catch

```
try { }
catch (xxx e) { }
catch (yyy e) { }
```

(Valid)

Case 10 multiple catch

```
try { }
catch (xxx e)
{
}
Catch (xxx e)
{
}
```

(Not valid)

Case 11

multiple Exceptions in one catch{}

```
try { }
Catch ( xxx | yyy e )
{
}
```

(Valid)

Case 12 try-catch-finally

```
try
{
}
catch (xxx e) { }
finally { }
```

(Valid)

Case 13

~~try~~ - finally

```
Catch
catch (xxx e) { }
finally { }
```

(Not valid)

Case 14 try-finally

```
try { }
finally { }
```

(Valid)

Case 15 unordered

```
try { }
finally { }
catch (xxx e) {
}
```

(Not valid)

Case 16

try multiple
Catch-finally

```
try {}
Catch (xxx e)
 {}
Catch (xxx e) {}
finally {}
```

(Valid)

Case 17 multiple finally

```
try { }
Catch (xxx e)
 { }
finally { }
 finally { }
```

(Not Valid)

Case 18 : Statements in-blw -try-catch-finally

```
try { }
 System.out.println ("Hi");
 Catch (xxx e) { }
 System.out.println ("Hello");
 finally { }
```

(Not Valid)

Case 19 : only try with resource

try (R) {} Valid

---

Case 24 - Nested try-catch-finally

```
try {
  try{}
 catch(xxxee){}
 finally {}
}
```

```
catch (xxx e){
  try{}
 Catch (xxx e){}
 finally {}
}
```

```
finally {
  try{ }
 catch (xxx e) { }
  finally { }
}
```

## Synchronous Exceptions

Occurs at a Specific program statement.

```
class lanuch {
  public static void main(String[] args)
  {
    String str = null;
    Str.toUpperCase();
  }
}
```

output

Java.lang.Nullpointer Exception
at launch.main(Lauch.java:6)

---

```
class launch {
  public static void main(String[] args)
  {
    int [] a = new int [5];
    a[5] = 10;
  }
}
```

output

Java.lang.ArrayIndexOutBounds
exception: at launch.main(launch
.java:6)

---

## Asynchronous Exceptions

occurs anywhere in the program.

```
class launch
{
  public static void main (String[] args)
  {
    Scanner scan = new Scanner(System.in);
    String name = scan.next();
    String grade = scan.next();
  }
}
```

Output

Enter your name:
Sachin
Enter your grades:
Ctrl+C —— keyboard Interrupt
Exception in thread "main" terminate batch job (Y/N)!

(Examples keyboard interrupts)

←——o——

xxx

Custome Exception

↓

User defined exception

parent class of all classes

```
class throwable extends Objects
{
  ==
  string msg;  (Constructr)  — Specialized setter
  public throwable(string msg) {
    this.msg = msg;
  } } — get.message() — one method (hirarcy)
class Exception Extends throwable.
      Exception (string msg)
       {
        Super (msg);
       }
  }
  class ICE extends exception. {
      ICE (String msg) (
       Super (msg);
  }
}
```

while creating obj passing String ( )