Java 8 Features

→ it come in march 2014
→ it enable functional programming (i.e → expr)
so we can write concise code (less)

★ for returning a square of number instead of writing function,
we can do it
⇒ import Java.util.function.*;
Function<Integer, Integer> f = i → i*i;
input / output para
S.O.p(f.apply(5));

★ Features :-

1] Lambda express
2] functional Interface (is a Ananoyms function) i-e nameless
3] Default mtds & static mthds
4] Predefined functional Interface

★ To write lambda expression
1] No name
2] No Modifiers
3] No return type

{} ⇒ optional for multiple line use {}

Public void m1(){ ⇒ () → { Sop("Hello"); }
S.O.p("Hello"); }
use Arrow operator

★ () → Sop("Hello");
⇒ m1(int a, int b) { Sop(a+b); }
(int a, int b) → { Sop(a+b); }
★★★ ⇒ Compiler can Automaily guess the parameter
in curly {} we should write return. it's else not necessary

FI is contain only SAM, To Apply expression we need FI

* To call we need FI (functional Interface)

Ex:-

Comparable → CompareTo()

Comparator → compare()

Runnable → run()

=> the SAM should be there to use FI

* functional Interface should contain only exactly one (SAM) & any number of static & default methods.

use Annotation => @FunctionalInterface

@fi in a { m1(); }

@ intf b extends a { m1() }

1) if b in

```
@FunctionalInterface
inteface inf {
    Public void m1();
}
```

* No class file is generated for lambda expression → treated as normal private method.

* We can use multithreading concept using lambda expression we write

```
X. class t implaments Runnable { public void run() { sop("Hi"); };
   class Test {
     main () {
   X t tr=new t();
     Thred m=new Thread(tr)
       tr.Start(); } }
```

=>

```
// we can write using λ expr
Runnable r= () -> {
        sop("child "); };
Thread t=new Thread(r);
   t.Start();
```

for Comparator we concise

```
Comparator<Integer> c=(I1,I2) -> (I1<I2)?-1:(I1>I2)?1:0;
   Collections.sort(l, C);
```

4. Emp In interface we cannot use object class method as default mthd because they are directly available to the class // we cannot use the default keyword for implemented class are public keyword

* Anonymous Inner class

```
Thread t=new Thread();
{ ... };
```

Writing class that extends thread class

for interface

```
Runnable r=new Runnable(){ ... };
```

Not to provide semicolon after Anonym

* Anonymous inner class is more-powerfull than lambda Expression

```
Interface A {
  m1();
  m2(); }
```

```
A a=new A()
{ public void m1() { ... }
  Public void m2() { } };
```

We can write the Anonymous class but not λ expression.

* if Anonymous Inner class contains single Abstract method than we can Replace with lambda expression

* from 1.8 version :- (also known as Defender method / function)
→ we can write default & Static methods in the interface & from 1.9v we can write private method

⇒ Suppose a Interface has 2 mthd so the implemented class should provide mthd definition, & 10 class have implemented the Interface & if we add one more method to Interface than the 10 classes should also provide the implen so to fix this the default method is added to the 1.8v

Suppose we are using multiple inheritence using Interface, than the both should not have same default method, & if default method then provide its implementation in implemented class.

⇒ to call the interface method in the implemented class, use interface_name.super.method_name();

★ From 1.8v interface can also have static Methods as well should use it for the class as it is more costly & the static method is no way related to the object & class is a heavy implementation & it has constructor, object

★ To call static method compulsory use the interface name

★ we can also write psvm (main mthd) in Interface & call directly.

★ Static methods are not by default available to implementor classes, we need to call them using interface name.

★ Predifined Fanctional Interface :-

① Predicate<T>; ⇒ whenever we want to check, boolean condition, only one method public boolean test(Integer I);

Syntax:- Predicate<String> P=S1->S1.length()>5;
S.O.P(P.test("Mayuresh")); //true

we can use and(), or(), negate() with Predicate

Perdicate<Integer> P1=i->i%2==0;
Perdicate<Integer> P2=i->i>10;
S.O.P(P1.~~test(10)~~.and(P2).test(12));
P1.or(P2).test(12));
P1.negate().test(12);

Function<T,R>

★ Function concept ⇒ When we have to get a different o/p from a functional Interface then use function concept.

```
Function<String, Integer> F = i -> i.length();
                  I/p   o/p
S.O.p(f.apply("mayur"); //5
```

★ We can also compose the function :- using

```
64 // f1.andThen(f2).apply(2); /// here first f1 is evaluated & then f2
16 // f1.compose(f2).apply(2); // here f2 is executed & then f1
```

```
Function<Integer> f1 = i -> 2*i;
Function<Integer> f2 = i -> i*i*i;
```

★ Consumer ⇒ When we dont have to return any value, then we use consumer it has no return types & no operation to perform

```
Consider<Student> C = s -> { S.O.p(s.marks); };
C.accept(s);
```

★ Supplier ⇒ Suppose we dont give any i/p but always need some object or value as o/p then we use Supplier.

```
S.get();
```

★ BiPredicate<Integer, Integer> p = (a,b) -> (a+b)%2==0;

```
S.O.p(p.test(10,20)); //true
```

When we need to check 2 args as a i/p parameter

⇒ We can use IntToDoubleFunction f = i → Math.sqrt(i

So no internal autoboxing & autounboxing

★ for BiFunction<Integer, Integer> f = (a, b) → i * i;

f.apply(7);

→ Int →

→ Integer → Int

→ Integer

★ We can use method reference for code reusability instead of writing lambda expression

"::" using double colon

```
Class Test
{
  Public Static void m1() { some code }
  PSVM(String[] args) {
    Runnable r = () -> { some code }; // lambda expression
    Runnable r = Test::m1; // Method refrence
    Test t = new Test(r);
    t.start();
  }
}
```

Here run() is referencing class method.

★ We can reuse already existing method instead of writing lambda expression.

★ We can use Static & instance method as reference.

- if Static method use ⇒ classname::method_name;
- if instance method use ⇒ Test t = new Test(); t::m1; without brackets

★ argument must be matched // return type can be differ

★ if Implementation is already available then go for method reference else not available use lambda expression.

Java.lang Package

JSPM
Page No

Equals ⇒ it is method which checks the actual content of the object

Example → Stud s1=new stud("1");
Stud s2=new stud("2");
S.equals(s2); //return false ⇒ it checks the obj. internally invokes the object class memory location "==" method which checks memory location.

★ Hashcode() ⇒ to generate a address, to store the value.

★ Java.lang. Packages:-

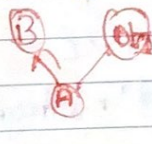
⇒ To write most common programs the most Package that required to run Program are encapsulated in Java.lang Package.

★ for Simple ① class Test { ②
public s.um(str[]) {
s.o.p("Hello"); } }
③ ④ → All are dependent on lang Pckg

★ object class act as the parent class, as it has most commonly used method, like hashCode(), equals(), for every predefined or customize class

Class A extends B { }

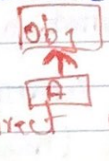
not Multiple ✗



✓ multilevel inheri

A as it is indirect child of object

Class A { }



Direct child of Object

Object class contains 11 methods

1) public String toString()
2) public native int hashCode()
3) boolean equals(Object o)
4) protected Object clone() throws CloneNotSupportedException
5) protected void finalize() throws Throwable
6) public void wait() throws InterruptedExc
7) public void wait(long ms) throw IE
8) public void wait(long ms, int ns) throws IE
9) public final void notify()
10) public final void notifyAll()
11) public final class getClass()

```
Psvm()
{
Class c = Class.forName("java.lang.Object");
Method[] m = c.getDeclaredMethods();
for(Method m1:m)
  {S.o.p("The name is"+m1.getName());}
```

* Strictly obj class has 12 mthds, i.e registerNatives but not required to us, it is for internal use.

① **toString()** :- To get the object representation of toString() is used, when we print any object reference toString() is called.

```
String s=obj.toString();
```

=> if our class does not contain toString() then obj class toStr() is called.

```
Student S=new Student();
Student@1784
```

=> Parent class i.e obj toString() called.

its implementation is => Public String toString(){

```
return getClass().getName()+"@"+Integer.toHexString(hashCode());
```

* In our code always override toString() & all wrapper class have their toString() => String, Integer, Float, Collections

Object class

hashCode() ⇒ In java every object has a unique number associated to it, so the search operation becomes easy (not address) for storing in Hashing related DS like. HashTable HashMap, HashSet, etc.

* Time complexity ⇒ O(1)
* HashCode is counted using Memory Address
→ We can override hashCode to generate our own number. (it must be unique number)

* if toString() of object class is called it internally invoke hashCode()
→ if we write our own hashCode() then it may or may not call hashCode().

⇒ if we override hashCode() but not toString() in our derived class,
→ than the toString() of obj class is called which call's in format ⇒ class_Name@ our hashCode() value
⇒ return Integer.toHexString(hashCode()); // Hex number

⇒ if we override toString() & hashCode() ⇒ direct call to toString() no hashCode() is called.

equals() ⇒ to check equality of two objects.
* if our class does not contain equals() then obj class method is called. it checks for the object reference and not contains.

```
S1=new C1();
S2=new C1();
S3=new C10();
S1=S4
S1.eq(S2) //false
S2.eq(S3) //false
S1.eq(S4) //true
S1.eq("mayur") //false
```

for overridding equals() → understand following:-
if we are passing diff type of obj, it should not return ClassCastException, we need to handle using try{ } catch(ClassCastException e) { return false; }

S1.equals(null); // if two obj are equal to "==" then both obj are also o1.equals(o2)

=> if we have null to compare then we need to handle NPE

```
try { } catch (NPE e) { return false; };
```

```
Public boolean equals(Object obj) {
try { int roll = this.roll;
      int name = this.name;
      Student s = (Student) obj;
      int r2 = s.roll;
      int name2 = s.name;
      if (name.equals(name2) && roll == r2)
        { return true; }
      else { return false; }
    } catch (ClassCastException e) { return false; }
    catch (NPE e) { return false; }
```

=> in instance method => Student s = new Stud(1, "Abc");

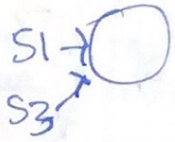
```
Public boolean equals() {
```

☆ int rollno = this.rollno is same also int rollno = rollno;

```
if (roll == s.roll && name.equals(s.name)) { }
```

=>

```
Public boolean equals(Object obj) {
  if (obj instanceof Student) {
    Student s = (Student) obj;
    if (name.equals(s.name) && roll == s.roll) {
      return true; } else
    { return false; }
  }
  return false; }
```



String class has equals overridden for content comparision but StringBuffer does not override equals method

☆ Suppose we have S1 & S3 pointing to same Object then no need to perform equals(), Add following to first line.

```
if (obj == this) { return true; } // Both pointing to same obj
```

equals() and "==" operator

1. r1 == r2 then r1.equals(r2) // true
2. r1 != r2 then r1.equals(r2) // may be
3. r1.equals(r2) then r1 == r2 // may or may not

=> if different types are compared then return false

- if the object are stored in same bucket in the Hashtable, then the object may or may not be same. even if the Hashcode is same.

- relation between equals() & hashcode();

=> if 2 obj are equal by .equals with then their Hashcode must be same.
=> that means 2 equivalent obj have same hashcode i.e r1.equals(r2) then r1.hashcode() == r2.hashcode() is true.

- Object class equals() & hashcode() method follow below contract, i.e whenever we are overridding Equals we should compulsory override the hashcode(). To satisfy above contract. i.e 2 obj should have same hashcode())

- if 2 obj not equal by equals() then their no restric on Hashcode may be equal or may not be equal.
- if Hashcode are equals then may be or not same

* if hashCode() of 2 objs are not equal then is obj are not equal by equals().

* In String class equals() overridden for content comparis & HashCode() is overridden to generate hashCode based on content

* In StringBuffer equals() not overriden for content compar & HashCode() is also not overridden

Note

* On which parameter we override the equals() we should use those parameter in hashCode(), it is a good Programming Practice.

* In all wrapper classes, collection class, String class equals() is overridden for content comparision Hence it is Highly recommended to override content comparision.