# JAVA 8 NEW FEATURES

MAYUR CHOURASIYA

## Contents

- Lambda Expressions
- Functional Interface
- Default Methods in Interface
- Static Methods in Interface
- Predefined functional Interface
  - Predicate
  - Function
  - Consumer
  - Supplier
- Method reference and constructor reference by (::) operator
- Stream API
- Date and Time API (joda.org)

## Lambda Expression

- **Lambda Expression is just an anonymous (nameless) function. That means the function which doesn't have the name, return type and access modifiers.**

```
Ex: 1                                            λ - Expression
public void m1() { sop("hello"); }  =>        () -> { sop("hello"); }
                                        OR    () -> sop("hello")
Ex : 2
Public void add(int a, int b)       =>        (int a,int b) -> { sop(a+b); }
{ sop(a+b); }                           OR    (a,b) -> sop(a+b);

Ex: 3
public String str(String s)        =>         (String s) -> { return s.length(); }
{ return s.length(); }                  OR    (s) -> return s.length();
                                        OR     s  -> s.length();
```

- **The Main Objective of Lambda Expression is to bring benefits of functional programming into Java.**

## Functional Interface

- If an interface contain only one abstract method, such type of interfaces are called functional interfaces and the method is called functional method or single abstract method (SAM).

  Ex:

  |  |  |
  |---|---|
  | 1) Runnable | : It contains only run() method |
  | 2) Comparable | : It contains only compareTo() method |
  | 3) ActionListener | : It contains only actionPerformed() method |
  | 4) Callable | : It contains only call() method |

- Inside functional interface in addition to single Abstract method (SAM) we write any number of default and static methods.

## Example of functional interface

Ex 1 :

```
Interface Interf {
    public void m1();
    default void m2() {
System.out.println(“hello”);   }
}
```

**In Java 8, Sun Micro System introduced @Functional Interface annotation to specify that the interface is Functional Interface.**

**Inside Functional Interface we have to take exactly only one abstract method.If we are not declaring that abstract method then compiler gives an error message.**

**Ex 2 :**

```
@Functional Interface
Interface Interf
{
   public void m1();
}
```

## Functional Interface vs Lambda Expression

Once we write Lambda expressions to invoke it's functionality, then Functional Interface is required.
We can use Functional Interface reference to refer Lambda Expression.

*Without Lambda Expression*

```
interface Interf {
    public void sum(int a,int b);
}

class Demo implements Interf {
    public void sum(int a,int b) {
        System.out.println("The 
sum:"•+(a+b));
    }
}

public class Test {
    public static void 
main(String[] args) {
        Interf i = new Demo();
        i.sum(20,5);
    }
}
```

*With Lambda Expression*

```
interface Interf {
    public void sum(int a, int b);
}

class Test {
    public static void main(String[] args) {
        Interf i = (a,b) -> System.out.println("The 
Sum:"+(a+b));
        i.sum(5,10);
    }
}
```

# Conclusion for functional Interface

- It should contain only one abstract method.
- It can contain any number of static and default method.
- It acts as a type for lambda expression.

  Ex :

  ```
  Interf i = () -> sop("hello");
  ```
- It can be used to invoke lambda expression.

- Lambda Expressions with Collections
  - Sorting elements of ArrayList using Lambda Expression
  - Sorting elements of TreeSet using Lambda Expression
  - Sorting elements of TreeMap using Lambda Expression

## Sorting elements of ArrayList using Lambda Expression

### Without using lambda expression

```
Class MyComparator implements
Comparator<Integer>
{
    Public int compare(Integer a, Integer
b) { return a-b; }
}

Class Test
{
    Public Static void main(String[] args)
    {
        ArrayList<Integer> l = new
ArrayList<Integer>();
        l.add(7);
        l.add(3);
        l.add(4);
        Cllections.sort(l, MyComparator());
        System.out.println("After
sorting"+l)
    }
}
```

```
Output : After sorting : [ 3 4 7
```

### With using lambda expression

```
Class Test
{
    Public Static void main(String[] args)
    {
        ArrayList<Integer> l = new
ArrayList<Integer>();
        l.add(7);
        l.add(3);
        l.add(4);

        Cllections.sort(l, (a,b) -> a-b; );
        System.out.println("After sorting"+l)
    }
}
```

## Sorting elements of Treeset with Lambda Expression

Ex :

```
TreeSet<Integer> map = new TreeSet<Integer>((a,b) -> (a>b)?-1:
(a<b)?1 : 0);
```

## Sorting elements of TreeMap with Lambda Expression

Ex :

```
TreeMap<Integer,String> map = new TreeMap<Integer,String>((a,b) ->
(a>b)?-1: (a<b)?1 : 0);
```

# Anonymous Inner Class vs Lambda Expression

**Wherever we are using anonymous inner classes there may be a chance of using Lambda expression to reduce length of the code and to resolve complexity.**

- Ex: With anonymous inner class

```
class Test {
    public static void main(String[]
args) {
        Runnable r = new Runnable() {
            public void run() {
                for(int i=0; i<10; i++) {

System.out.println("Child Thread");
                }
            }
            Thread t = new Thread(r);
                t.start();
                for(int i=0; i<10; i++)

System.out.println("Main thread");
        }
    }
```

- With Lambda expression

```
class Test {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            for(int i=0; i<10; i++) {
                System.out.println("Child
Thread");
            }
        });
        t.start();
        for(int i=0; i<10; i++) {
            System.out.println("Main Thread");
        }
    }
}
```

## Advantages of Lambda expression

- We can reduce length of the code so that readability of the code will be improved.
- We can resolve complexity of anonymous inner classes.
- We can provide Lambda expression in the place of object.
- We can pass lambda expression as argument to methods.

### Note :

- Inside lambda expression we can’t declare instance variables.
- Whatever the variables declare inside lambda expression are simply acts as local variables
- Within lambda expression ‘this” keyword represents current outer class object reference (that is current enclosing class reference in which we declare lambda expression)

## Default Methods in Interfaces

- Until 1.7 version onwards inside interface we can take only public abstract methods and public static final variables (every method present inside interface is always public and abstract whether we are declaring or not).
- Every variable declared inside interface is always public static final whether we are declaring or not.
- But from 1.8 version onwards in addition to these, we can declare default concrete methods also inside interface, which are also known as defender methods.
- The main advantage of default methods is without effecting implementation classes we can add new functionality to the interface (backward compatibility).

Ex :

```
interface Interf {
    default void m1() {
        System.out.println("Default Method");
    }
}

class Test implements Interf {
    public static void main(String[] args) {
        Test t = new Test();
        t.m1();
    }
}
```

## Static Methods in Interfaces

- From 1.8 version onwards in addition to default methods we can write static methods also inside interface to define utility functions.
- Interface static methods by-default not available to the implementation classes hence by using implementation class reference we can't call interface static methods. We should call interface static methods by using interface name only.
- As interface static methods by default not available to the implementation class, overriding concept is not applicable.
- Based on our requirement we can define exactly same method in the implementation class, it's valid but not overriding.

Ex:

```java
interface Interf {
    public static void sum(int a, int b) {
        System.out.println("The Sum:"+(a+b));
    }
}

class Test implements Interf {
    public static void main(String[] args) {
        Test t = new Test();
        t.sum(10, 20);              //Compilation Error
        Test.sum(10, 20);           //Compilation Error
        Interf.sum(10, 20);         //Works fine, Static method can be called using Interface name only
    }
}
```

## Predefined functional Interface : Predicates

- A predicate is a function with a single argument and returns boolean value.
- To implement predicate functions in Java, Oracle people introduced Predicate interface in 1.8 version (i.e.,Predicate<T>).
- Predicate interface present in *Java.util.function* package.
- It's a functional interface and it contains only one method i.e., test()

Ex:

```
interface Predicate<T> {
    public boolean test(T t);
}
```

- As predicate is a functional interface and hence it can refers lambda expression

**Ex:1 Write a predicate to check whether the given integer is greater than 10 or not.**

```
public boolean test(Integer I) {
if (I >10) { return true; } else { return false; }}
```

Without Lambda Expression

```
(Integer I) -> { if (I >10) { return true; } else {
return false; } }
```

With Lambda Expression

```
                I -> (I>10);    (Lambda Expression shortest form)
```

```
     predicate<Integer> p = I -> (I >10);
             System.out.println (p.test(100));     true
             System.out.println (p.test(7));       false
```

## Program:

```
import Java.util.function;
   class Test {
       public static void main(String[] args) {
           predicate<Integer> p = I -> (I>10);
           System.out.println(p.test(100));  //true
           System.out.println(p.test(7));    //false
           System.out.println(p.test(true)); // CE
       }
   }
```

## Predicate joining

It’s possible to join predicates into a single predicate by using the following methods.

and()

or()

negate()

these are exactly same as logical AND ,OR complement operators

## Example of Predicate join

```
import Java.util.function.*;
   class test {
       public static void main(string[] args) {
           int[] x = {0, 5, 10, 15, 20, 25, 30};
           predicate<integer> p1 =  i -> i>10;
           predicate<integer> p2 =  i -> i%2==0;
           System.out.println("The Numbers Not Greater Than 10:");
                   m1(p1.negate(), x);
           System.out.println("The Numbers Greater Than 10 And Even Are:â€•);
                   m1(p1.and(p2), x);
           System.out.println("The Numbers Greater Than 10 OR Even:â€•);
                   m1(p1.or(p2), x);
       }

       public static void m1(predicate<integer> p, int[] x) {
           for(int x1 : x) {
               if(p.test(x1))
                   System.out.println(x1);
           }
       }
   }
```

## Predefined functional Interface : Function

- Functions are exactly same as predicates except that functions can return any type of result but function should (can) return only one value and that value can be any type as per our requirement.
- To implement functions oracle people introduced Function interface in 1.8version.
- Function interface present in Java.util.function package.
- Functional interface contains only one method i.e., apply()

- Ex :

```
interface function(T,R) {
    public R apply(T t);
}
```

## Assignment: Write a function to find length of given input string.

Ex:

```
class Test {
    public static void main(String[] args) {
        Function<String, Integer> f = s ->s.length();
        System.out.println(f.apply("Durga"));
        System.out.println(f.apply("Soft"));
    }
}
```

## Function Chaining

We can combine multiple functions together to form more complex functions.For this Function

interface defines the following 2 default methods:

f1.andThen(f2) : First f1 will be applied and then for the result f2 will be applied.

f1.compose(f2) : First f2 will be applied and then for the result f1 will be applied.

## Predefined functional Interface : Consumer

Sometimes our requirement is we have to provide some input value, perform certain operation, but

not required to return anything , then we should go for Consumer i.e. Consumer can be used to

consume object and perform certain operation.

Consumer Functional Interface contains one abstract method accept.

Ex:

```
 interface Consumer<T>
 {
 public void accept(T t);
 }
```

## Consumer Chaining

- Just like Predicate Chaining and Function Chaining, Consumer Chaining is also possible. For this
- Consumer Functional Interface contains default method andThen().
- Structure : **c1.andThen(c2).andThen(c3).accept(s)**
- First Consumer c1 will be applied followed by c2 and c3.

- Ex :

```
public void authenticateUser()
{
    Consumer<Student> c1 = s -> System.out.println("Username check" + s.getUserName());
    Consumer<Student> c2 = s -> System.out.println("Password check" + s.getPwd());
    Consumer<Student> c3 = s -> System.out.println("Valid, Allow login to" + s.getName());

    Student s = new Student("Mayur", "MG2333", "JHBDBU@123");
    Consumer<Student> chainedConsumer = c1.andThen(c2).andThen(c3);
    chainedConsumer.accept(s);
}
```

# Predefined functional Interface : Supplier

Sometimes our requirement is we have to get some value based on some operation like supply Student object, Supply Random Name etc.

For this type of requirements we should go for Supplier.

Supplier can be used to supply items (objects).

Supplier won't take any input and it will always supply objects.

Supplier Functional Interface contains only one method get()

Ex :

```
interface Supplier<R>
{
public R get();
}
```

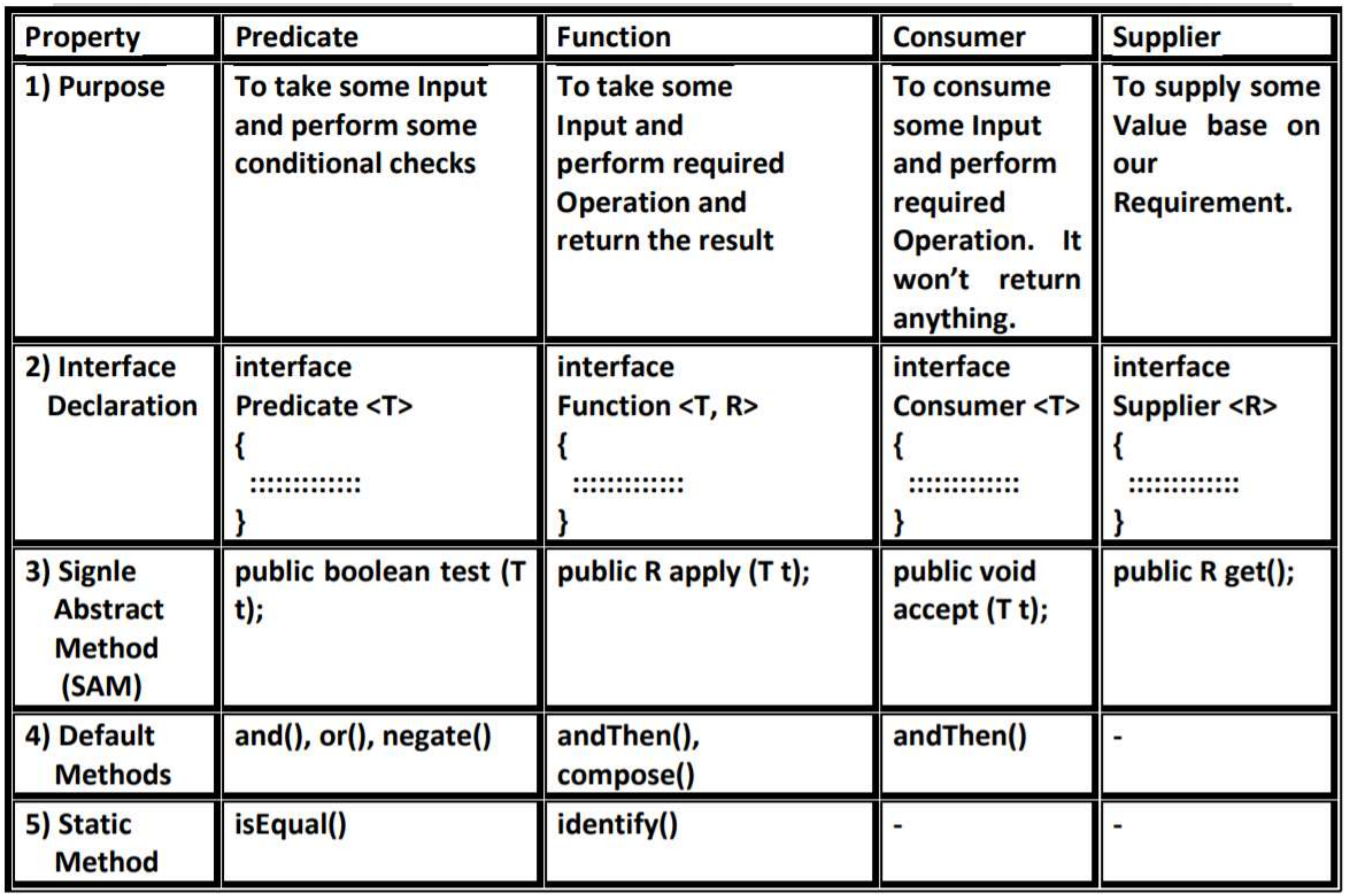
Supplier Functional interface does not contain any default and static methods.

## Program : Write a supplier to supply System Date

```
public static void main(String[] args) {
    Supplier<Date> d = () -> new Date();
    System.out.println(d.get());

}
```

## Comparison Table of Predicate, Function, Consumer and Supplier



| Property | Predicate | Function | Consumer | Supplier |
|---|---|---|---|---|
| 1) Purpose | To take some Input and perform some conditional checks | To take some Input and perform required Operation and return the result | To consume some Input and perform required Operation. It won't return anything. | To supply some Value base on our Requirement. |
| 2) Interface Declaration | interface Predicate <T> { :::::::::::: } | interface Function <T, R> { :::::::::::: } | interface Consumer <T> { :::::::::::: } | interface Supplier <R> { :::::::::::: } |
| 3) Signle Abstract Method (SAM) | public boolean test (T t); | public R apply (T t); | public void accept (T t); | public R get(); |
| 4) Default Methods | and(), or(), negate() | andThen(), compose() | andThen() | - |
| 5) Static Method | isEqual() | identify() | - | - |

# Streams

- To process objects of the collection, in 1.8 version Streams concept introduced.

## Without Streams

```
public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<>();
    list.add(1); list.add(2); list.add(6); list.add(5);

    ArrayList<Integer>  list2 = new ArrayList<>();
    for (int e : list)
    {
        if (e%2==0) {
            list2.add(e);
        }
    }
    System.out.println(list2);
}
```

## With Streams

```
public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<>();
    list.add(1); list.add(2); list.add(6); list.add(5);

    List<Integer> list2 = list.stream().filter(k->k%2==0).collect(Collectors.toList());
    System.out.println(list2);
}
```

## STREAMS API

- To process objects of the collection, in 1.8 version Streams concept introduced.
- We can create a stream object to the collection by using stream() method of Collection interface. stream() method is a default method added to the Collection in 1.8 version.

  default Stream stream()

  Ex: Stream s = c.stream();
- Stream is an interface present in java.util.stream. Once we got the stream, by using that we can process objects of that collection.
- We can process the objects in the following 2 phases

  1.Configuration

  1.1 Filter

  1.2 Map

  2.Processing

## 1) Configuration:

**We can configure either by using filter mechanism or by using map mechanism**

### 1.1 Filtering

- We can configure a filter to filter elements from the collection based on some boolean condition by using filter()method of Stream interface.

  ```
  public Stream filter(Predicate<T> t)
  ```

- here (Predicate<T > t ) can be a boolean valued function/lambda expression

  Ex:

  ```
  Stream s = c.stream();
  Stream s1 = s.filter(i -> i%2==0);
  ```

- Hence to filter elements of collection based on some Boolean condition we should go for filter() method.

## 1.2 Mapping

- If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for map() method of Stream interface.

  public Stream map (Function f);

- It can be lambda expression also

  Ex:

  ```
  Stream s = c.stream();
  Stream s1 = s.map(i-> i+10);
  ```

- Once we performed configuration we can process objects by using several methods.

## 2) Processing

- processing by collect() method
- Processing by count()method
- Processing by sorted()method
- Processing by min() and max() methods
- forEach() method
- toArray() method
- Stream.of()method

## Processing by collect() method

This method collects the elements from the stream and adding to the specified to the collection indicated (specified) by argument.

```
List<Integer> l2 = l1.stream().filter(i -> i%2==0).collect(Collectors.toList());
```

## Processing by count() method

This method returns number of elements present in the stream.

Method : public long count()

```
long count = l1.stream().filter(i -> i%2==0).count();
```

## Processing by sorted()method

If we sort the elements present inside stream then we should go for sorted() method.

the sorting can either default natural sorting order or customized sorting order specified by comparator.

sorted()- default natural sorting order

sorted(Comparator c)-customized sorting order.

```
Ex :
List<String> l= list.stream().sorted().collect(Collectors.toList());

List<String> l=l.stream().sorted((s1,s2) -> s1.compareTo(s2)).collect(Collectors.toList());
```

## Processing by min() and max() methods

min(Comparator c)

returns minimum value according to specified comparator.

max(Comparator c)

returns maximum value according to specified comparator

```
EX : String max = l.stream().max((s1,s2) ->
s1.compareTo(s2)).get();
```

## forEach() method

This method will not return anything.

This method will take lambda expression as argument and apply that lambda expression for each element present in the stream.

```
EX :  l.stream().forEach( s-> sop(s) );
      l3.stream().forEach(System.out:: println);
```

## toArray() method

We can use toArray() method to copy elements present in the stream into specified array.

Ex :

```
Integer[] ir = l1.stream().toArray(Integer[] :: new);
    for(Integer i: ir) {
        sop(i);
      }
```

## Stream.of() method

We can also apply a stream for group of values and for arrays.

Ex:

```
Stream s=Stream.of(99,999,9999,99999);
s.forEach(System.out:: println);


Double[] d={10.0,10.1,10.2,10.3};
Stream s1=Stream.of(d);
s1.forEach(System.out :: println);
```

## Method reference by (::) operator

- Functional Interface method can be mapped to our specified method by using :: (double colon) operator. This is called method reference.
- Our specified method can be either static method or instance method.
- Functional Interface method and our specified method should have same argument types, except this the remaining things like return type, method name, modifiers etc. are not required to match.

Syntax:

- If our specified method is static method

**Classname::methodName**

- If the method is instance method

**Objref::methodName**

- Functional Interface can refer lambda expression and Functional Interface can also refer method reference. Hence lambda expression can be replaced with method reference. Hence method reference is alternative syntax to lambda expression.
- In the below example Runnable interface run() method referring to Test class static method m1().

Ex: With Lambda Expression

```
Class Test {
    public static void main(String[]
args) {
        Runnable r = () □ {
            for(int i=0; i<=10; i++) {
System.out.println("Child Thread");
            }
        };
        Thread t = new Thread(r);
        t.start();
        for(int i=0; i<=10; i++) {
            System.out.println("Main
Thread");
        }
    }
}
```

With Method Reference

```
Class Test {
    public static void m1() {
        for(int i=0; i<=10; i++) {
            System.out.println("Child
Thread");
        }
    }
    public static void main(String[]
args) {
        Runnable r = Test:: m1;
        Thread t = new Thread(r);
        t.start();
        for(int i=0; i<=10; i++) {
            System.out.println("Main
Thread");
        }
    }
```

- The main advantage of method reference is we can use already existing code to implement functional interfaces (code reusability).
- In the below example functional interface method m1() referring to Test class instance method m2().
- Ex :

```
interface Interf {
    public void m1(int i);
}
class Test {
    public void m2(int i) {
        System.out.println("From Method Reference:"+i);
    }
    public static void main(String[] args) {
        Interf f = I ->sop("From Lambda 
Expression:"+i);
        f.m1(10);
        Test t = new Test();
        Interf i1 = t::m2;
        i1.m1(20);
    }
}
```

## Constructor References by (::) operator

- We can use :: ( double colon )operator to refer constructors also
- Syntax: classname :: new

```
class Sample {
    private String s;
    Sample(String s) {
        this.s = s;
        System.out.println("Constructor Executed:"+s);
    }
}
interface Interf {
    public Sample get(String s);
}
class Test {
    public static void main(String[] args) {
        Interf f = s -> new Sample(s);
        f.get("From Lambda Expression");
        Interf f1 = Sample :: new;
        f1.get("From Constructor Reference");
    }
}
```

Note: In method and constructor references compulsory the argument types must be matched.

## Date and Time API: (Joda-Time API)

- Until Java 1.7 version the classes present in Java.util package to handle Date and Time (like Date, Calendar, TimeZoneetc) are not up to the mark with respect to convenience and performance.
- To overcome this problem in the 1.8version oracle people introduced Joda-Time API. This API developed by joda.org and available in Java in the form of Java.time package.
- program for to display System Date and time.

```
import Java.time.*;
   public class DateTime {
       public static void main(String[] args) {
           LocalDate date = LocalDate.now();
           System.out.println(date);
           LocalTime time=LocalTime.now();
           System.out.println(time);
       }
   }
```

O/p:

```
2015-11-23
12:39:26:587
```

- **Once we get LocalDate object we can call the following methods on that object to retrieve Day,month and year values separately.**

```
import Java.time.*;
class Test {
    public static void main(String[] args) {
        LocalDate date = LocalDate.now();
        System.out.println(date);
        int dd = date.getDayOfMonth();
        int mm = date.getMonthValue();
        int yy = date.getYear();
        System.out.println(dd+"..."+mm+"..."+yy);
        System.out.printf("\n%d-%d-%d",dd,mm,yy);
    }
}
```

- **Once we get LocalTime object we can call the following methods on that object.**

```
import java.time.*;
class Test {
    public static void main(String[] args) {
        LocalTime time = LocalTime.now();
        int h = time.getHour();
        int m = time.getMinute();
        int s = time.getSecond();
        int n = time.getNano();
        System.out.printf("\n%d:%d:%d:%d",h,m,s,n);
    }
}
```

- If we want to represent both Date and Time then we should go for LocalDateTime object.

  LocalDateTime dt = LocalDateTime.now();

  System.out.println(dt);

  O/p: 2015-11-23T12:57:24.531

## To Represent Zone

- ZoneId object can be used to represent Zone.

```
import Java.time.*;
class ProgramOne {
    public static void main(String[] args) {
        ZoneId zone = ZoneId.systemDefault();
        System.out.println(zone);
    }
}
```

- We can create ZoneId for a particular zone as follows

```
ZoneId la = ZoneId.of("America/Los_Angeles");
ZonedDateTimezt = ZonedDateTime.now(la);
System.out.println(zt);
```

## Period Object

- Period object can be used to represent quantity of time

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1989,06,15);
Period p = Period.between(birthday,today);
System.out.printf("age is %d year %d months %d
days",p.getYears(),p.getMonths(),p.getDays());
```

- write a program to check the given year is leap year or not

```
public class Leapyear {
    int n = Integer.parseInt(args[0]);
    Year y = Year.of(n);
      if(y.isLeap())
    {
        System.out.printf("%d is Leap year", n);
    } else {
        System.out.printf("%d is not Leap year",n);
    }
}
```

## References : Durga Soft (Udemy)