Telegram link(Class link and notes will be posted) ::
https://t.me/+DSb_Dc6n8BhkZTU1

What is the difference b/w Predicate(I) vs Function(I)
 => Predicate
      1. To implement conditional check.
      2. Predicate can take only argument of any Type.
            Predicate<T>
      3. Predicate defines only one method called test()
            public boolean test(T t)
      4. Predicate returns only boolean value

 => Function
      1. To perform some operation and return some result.
      2. Function can take 2 Parameters
            Function<T,R>
            T:: Input type
            R:: Output type
      3. Function defines onlyl one method called apply()
            public R apply(T t)
      4. Function can return any type of value.

Note:
   Lambda Expression would be used to refer to Functional Interface.
   We have alternative solution for Lambda Expression that is "Method and Constructor Reference".

Method and Constructor reference
++++++++++++++++++++++++++++++++
=> It can be used as an alternative to Lambda Expression.
=> Since it can be used as an alternative we can use this only for "Functional interface".
=> We can map functional interface methods through
      1. Lambda expression
      2. Method reference[userdefined methods like instance and static]

=> Functional interface method and our specific methods should have same argument types, except this remaining things like
            return-type, methodName,accessmodifier is not important.

Syntax for instance methods
      ojbName::methodName

Syntax for static methods
    ClassName::methodName

Logic using Lambda Expression
+++++++++++++++++++++++++++

```
Runnable r = ()->{
                  for(int i=1;i<=5;i++)
                  {
                        System.out.println("child thread");
                  }
           };
```

```
Logic using Method reference
++++++++++++++++++++++++++
public class Test{

      //Developer :: Karthik
      //UserDefined static method
      public static void logicForThread()
      {
                  //logic for child thread
                  for (int i =0;i<5 ;i++)
                  {
                        System.out.println("Child thread...");
                  }
      }

}

//binded to void run()
Runnable r1=Test::logicForThread;


One code can be written in mulitple forms
  a. Traditional Approach(OOPS)
      1. Ananymous inner class implements interface
      2. Ananymous inner class extends a class
      3. Ananmyous inner class passed as an argument.

 b. Functional Programming
     1. Lambda Expression[build the code from the scratch] for
Functionalinterface[SAM]
     2. MethodReference[Reuse the code] for Functional interface[SAM]


eg#1.
import java.util.function.*;

@FunctionalInterface
interface Interf
{
      public void m1(int i);
}

//logic is written by other developer
class Demo
{
      public int logicforReusing(int i)
      {
            System.out.println("Hey i gave the implementation.. reuse it ...");
            return 100;
      }
}

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

            //using lambda expression
            Interf i1 = i->System.out.println("coming from lambda
Expression..."+i);
            i1.m1(10);
```

```
            //using method reference
            Demo d1   = new Demo();
            Interf i2 = d1::logicforReusing;
            i2.m1(100);

      }
}
```

```
Output
coming from lambda Expression...10
Hey i gave the implementation.. reuse it ...
```

```
Constructor reference
++++++++++++++++++++
=> we use the same :: operator to refer constructor also.
=> Classname obj = new ClassName();//Traditional OOPS

Syntax
++++++
   ClassName :: new

eg#1.
import java.util.function.*;

@FunctionalInterface
interface Interf
{
      public Sample get(String s);
}

class Sample
{
      private String s;

      //Constructor
      Sample(String s)
      {
            this.s = s;
            System.out.println("Constructor Executed.... "+s);
      }
}

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

            //using lambda expression
            Interf i1= s->new Sample(s);
            i1.get("sachin :: using Lambda Expression");

            System.out.println();

            //using Constructor reference
            Interf i2 = Sample :: new ;
            i2.get("dhoni :: using Constructor Reference");
      }
}
```

```
Output
Constructor Executed.... sachin :: using Lambda Expression

Constructor Executed.... dhoni  :: using Constructor Reference



Optional API in JDK8
++++++++++++++++++++
 => It is given by Oracle to avoid NullPointerException in our program.
 => Normaly in realtime project every developer will not implement null checking in
the program, as a result of which there would
    be NullPointerException in the program.
 => To avoid this we use "Optional" API in realtime project.
 => It is always a good practise to keep the data which needs to be processed
inside "Optional" object only.


public final class java.util.Optional<T> {
  public static <T> java.util.Optional<T> ofNullable(T);
  public static <T> java.util.Optional<T> empty();
  public static <T> java.util.Optional<T> of(T);
  public T get();
  public boolean isPresent();
  public boolean isEmpty();

  public java.util.Optional<T> filter(java.util.function.Predicate<? super T>);
  public <U> java.util.Optional<U> map(java.util.function.Function<? super T, ?
extends U>);

}

eg#1.
import java.util.*;

import java.util.*;

//API Development
class User
{
      //Written by Developer without Optional API::Shahid
      public String getUserById(Integer id)
      {
            if(id == 10)
                  return "sachin";
            else if(id == 19)
                  return "dravid";
            else if(id == 7)
                  return "dhoni";
            else if(id == 18)
                  return "kohli";
            else
                  return null;
      }

      //Handling NullChecking through an API called "Optional"
      public Optional<String> getUserNameById(Integer id)
      {
            String name = null;
```

```
            if(id == 10)
                  name= "sachin";
            else if(id == 19)
                  name= "dravid";
            else if(id == 7)
                  name= "dhoni";
            else if(id == 18)
                  name= "kohli";

            //returns Non-empty Optional object if a value is present otherwise
return empty Optional object.
            return Optional.ofNullable(name);
      }
}
public class Test{
      public static void main(String[] args){

            //Written by Developer:: Karthik
            Scanner scanner = new Scanner(System.in);

            System.out.print("Enter the id :: ");
            Integer id = scanner.nextInt();

            User user = new User();
            /*
                  String userName=user.getUserById(id);
                  System.out.println("Hello :: "+ userName.toUpperCase());
            */

            Optional<String> optional=user.getUserNameById(id);
            if (optional.isPresent())
            {
                  String record = optional.get();
                  System.out.println("Hello :: " +record.toUpperCase());
            }
            else
            {
                  System.out.println("No Data found...");
            }
      }
}



Optional API methods
++++++++++++++++++++
public java.util.Optional<T> filter(java.util.function.Predicate<? super T>);
public <U> java.util.Optional<U> map(java.util.function.Function<? super T, ?
extends U>);

eg#1.
import java.util.*;
import java.util.function.*;

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

            //Creating an Optional Object with data
            Optional<String> nonEmptyGender=Optional.of("male");
```

```
            //Creating an Empty Optional Object
            Optional<String> emptyGender=Optional.empty();


            System.out.println("Lambda Expression :: "+nonEmptyGender.map(s-
>s.toUpperCase()));
            System.out.println("Method
Reference"+nonEmptyGender.map(String::toUpperCase));

            System.out.println();

            System.out.println("Lambda Expression :: "+ emptyGender.map(s-
>s.toUpperCase()));
            System.out.println("Method Reference ::
"+emptyGender.map(String::toUpperCase));

            System.out.println();

            Predicate<String> p1 = gender-> gender.equals("MALE");
            System.out.println(nonEmptyGender.filter(p1));
            System.out.println(nonEmptyGender.filter(gender->
gender.equals("male")));
            System.out.println(nonEmptyGender.filter(gender->
gender.equalsIgnoreCase("male")));

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

What is the difference b/w the follwing methods?
1. public boolean isPresent();
      It returns true if the given Optional object is non-empty(data present) otherwise it returns false.

2. public void ifPresent(java.util.function.Consumer<? super T>);

Consumer
+++++++
=> Sometimes our requirement is to provide some input value, perform certain operations but not requried to return anything.
=> Consumer can be used to consume the object and perform certain operation.

```
public interface java.util.function.Consumer<T> {
  public abstract void accept(T);
  public default java.util.function.Consumer<T>
andThen(java.util.function.Consumer<? super T>);
}

import java.util.*;
import java.util.function.*;

import java.util.function.*;

public class Test{
```

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

		//Binded to accept(T):: void of Consumer<T>
		Consumer<String> c=s->System.out.println(s);

		c.accept("sachin");
		c.accept("dravid");

		System.out.println();

		Consumer<String> c1 =System.out::println;
			c1.accept("shahid");
			c1.accept("karthik");
	}
}

Output
sachin
dravid

shahid
karthik


Requirement
++++++++++
Display movie information using Consumer interface

import java.util.*;
import java.util.function.*;

import java.util.function.*;

class Movie
{
	String name;
	String hero;
	String heroine;

	Movie(String name,String hero,String heroine)
	{
		this.name = name;
		this.hero = hero;
		this.heroine = heroine;
	}
}


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

		Movie[] movies = new Movie[4];
		addMovies(movies);

		Consumer<Movie> c = movie-> {
				System.out.println("Name    is :: "+movie.name);
				System.out.println("Hero    is :: "+movie.hero);
				System.out.println("Heroine is :: "+movie.heroine);
				System.out.println();
```

```
            };

            for (Movie movie: movies )
            {
                  c.accept(movie);
            }
      }
      public static void addMovies(Movie[] movies){
            Movie m1= new Movie("Salaar","Prabhas","ShruthiHasan");
            Movie m2= new Movie("Pushpa","AlluArjun","Rashmikha");
            Movie m3= new Movie("Sambahadhur","Vicky","katrinakaif");
            Movie m4= new Movie("Katera","Darshan","Nikitha");

            movies[0] = m1;
            movies[1] = m2;
            movies[2] = m3;
            movies[3] = m4;
      }
}

Output
Name    is :: Salaar
Hero    is :: Prabhas
Heroine is :: ShruthiHasan

Name    is :: Pushpa
Hero    is :: AlluArjun
Heroine is :: Rashmikha

Name    is :: Sambahadhur
Hero    is :: Vicky
Heroine is :: katrinakaif

Name    is :: Katera
Hero    is :: Darshan
Heroine is :: Nikitha



Write a program that student details as input print the student details along with
"Grade"
      Grade will be given based on the marks taken by the student
            marks>=80  [Distinction]
            marks>=60  [FirstClass]
            marks>=50  [SecondClass]
            marks>=35  [ThirdClass]

      Print only such students whose marks is greater than or equal to 65???


Printing         ----> Consumer  [accept(student)]
Grade            ----> Predicate [test(marks)]
Predicting Grade ----> Function<Student,String>::[apply]

eg#1.
import java.util.function.*;
class Student
{
      String name;
```

```
	int marks;

	Student(String name,int marks)
	{
		this.name  = name;
		this.marks = marks;
	}
}


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

		Student[] students = new Student[4];
		addStudents(students);

		Function<Student,String> f = s-> {
			int marks=s.marks;
			if (marks>=80)
				return "A[Distinction]";
			else if(marks>=60)
				return "B[FirstClass]";
			else if(marks>=50)
				return "C[SecondClass]";
			else if(marks>=35)
				return "D[ThirdClass]";
			else
				return "E[Failed]";
		};

		Consumer<Student> c = student-> {
				System.out.println("Name   is :: "+student.name);
				System.out.println("Marks  is :: "+student.marks);
				System.out.println("Grade  is :: "+f.apply(student));
				System.out.println();
		};

		Predicate<Student> p = s-> s.marks>=65;

		for (Student s :students )
		{
			if(p.test(s))
				c.accept(s);
		}
	}
	public static void addStudents(Student[] students){

		Student s1 =new Student("Shahid",80);
		Student s2 =new Student("Karthik",65);
		Student s3 =new Student("Nitin",55);
		Student s4 =new Student("Naveen",45);

		students[0]=s1;
		students[1]=s2;
		students[2]=s3;
		students[3]=s4;

	}
}
```

```
Output
Name   is :: Shahid
Marks  is :: 80
Grade  is :: A[Distinction]

Name   is :: Karthik
Marks  is :: 65
Grade  is :: B[FirstClass]
```