\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***Java 8 Observations**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

java.util.function pacakage added

->By using java 8 we can provide concise code

-> functional programming enabled

->Lambda Expressions (to enable Functional programming)

->Date and Time ApI's

->Default method and Static methods added

->Predefined functions(Function,Predicate,Consumer,Supplier...etc)

->:: operator--Method reference, Constructor reference

->Optional Classes

**=========================Functional==================**

Function<T,R>

ex-Function<Integer, Integer> value= a-> a\*a;

System.out.println(value.apply(2));//4

**Example:-**Predicate<Integer> intPrediacte=a->a\*a;

System.out.println( intPrediacte.test(4))//16

**========================Basic Lambda Example====**

//normal

public void sayHello(){

System.out.println("Hello This is uday");

}

**//after lambda**

()->{System.out.println("Hello This is uday"););

Examples for Functional Interfaces:-

Runnable,Comparable,Comparator,Callable

**============Default Method==========**

If any class is implementing Interface, then we should provide the implementation or we should declare it as a abstract.

if already so many classes are using this interface, we should o each and every place and need to implement

But by using default method it is optional

**Note:-**Object class methods cannot be added as default methods

(A default method cannot override a method from java.lang.Object )

**=======================Predefined Functional Interfaces=============**

->We can use it for conditional checks

**-\*\*\*\*\*\*Predicate\*\*\*\*\*\*\*:-**

only abstarct method is test()

default methods like

and() ,negate(),or()

ex:-

Predicate<Integer> intPredicate=a->a%2==0;

intPredicate.test(a)

**\*\*\*\*Function\*\*\*\*\*\*\*\*\*\***

f1.andThen(f2).apply(i); first f1 then f2

f1.compose(f2).apply(i); first f2 then f1

**==================Consumer===================**

**Consumer:-**Consumer is going to accept values but not return any thing

**methods:-**

accept(input);

Ex:-

Consumer<String> con=s->System.out.println(s);

con.accept("uday");

**==================Supplier=============================**

**Supplier:-** it is will not take any input but return value

methods:-

get();

ex:-

Supplier<Strnig> supplier=()->{

return "abc";

}

System.out.println(supplier.get());

**======================Two Argument Functional Interface===============**

**1.BiPredicate:-** 2 input args it will take

Normal Predicate take one argument and do some conditional check

BiPredicate will take 2 arguments and do some conditional check

**Example:-**

BiPredicate<Integer,Integer> biPredicate=(a,b)->a>b;

System.out.println(biPredicate.test(2, 5));

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**2.BiFunction:-**

**BiFunction<T1,T2,R>**

T1->Argument1

T2->Argument2

R->Return Type

**Exmaple:-**

List<Employee> list=new ArrayList<>();

BiFunction<Integer, String, Employee> biFunction=(eno,eName)->new Employee(eno, eName);

list.add(biFunction.apply(1, "uday"));

list.add(biFunction.apply(2, "kalluri"));

list.add(biFunction.apply(3, "saketh"));

list.add(biFunction.apply(2, "krishna"));

list.forEach(System.out::println);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**3.BiConsumer:-**

It will take 2 inputs but it will not return any thing .Will do some internal calculation

BiConsumer<Employee, Double> biConsumer=(e,number)->e.setSalary(e.getSalary()+number);

**Sample:-**

public class BiConsumerExample {

public static void main(String[] args) {

List<Employee> list=new ArrayList<>();

addEmployees(list);

BiConsumer<Employee, Double> biConsumer=(e,number)->e.setSalary(e.getSalary()+number);

for(Employee e:list) {

biConsumer.accept(e, 500.0);

}

list.forEach(System.out::println);

}

public static void addEmployees(List<Employee> list) {

list.add(new Employee(1, "uday",2000));

list.add(new Employee(2, "kalluri",3000));

list.add(new Employee(3, "saketh",40000));

list.add(new Employee(2, "krishna",5000));

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a problem if we are using Wrapper classes inside of Predicate.

There will be a unnecessary Boxing and Unboxing during the using of Predicate

for avoiding the above problems

InBuilt Predefined functions

Examples for Predefined Functional Interfaces examples for Primitive Datatypes:-

**1.IntPredicate:-**

IntPredicate intPredicate = i -> i > 5;

System.out.println(intPredicate.test(3));

**2.IntFunction:-**

IntFunction<Integer> val = i -> i \* 2;

System.out.println(val.apply(5));

**3.IntConsumer:-**

IntConsumer consumer = a -> System.out.println(a);

consumer.accept(5);

**4. IntSupplier:-**

IntSupplier sup=()->{

return 10;};

System.out.println(sup.getAsInt());

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Method Reference\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In case of method reference

Anonymous method and the custom method arguments should match you can ignore the Return type

**Static Method Reference:-**

public static void m1() {

for (int i = 0; i < 10; i++) {

System.out.println(i);

}

}

calling the above code in below code snippet:-

Runnable r = MethodReferanceExample::m1;

new Thread(r).start();

**Method Reference:-**

public void m1() {

for (int i = 0; i < 10; i++) {

System.out.println(i);

}

}

calling the above code in below code snippet:-

Runnable r = new MethodReferanceExample()::m1;

new Thread(r).start();

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Constructor Reference\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.zensar;

public class ConstructorReference {

public static void main(String[] args) {

Inter inter = Sam::new;

Sam s = inter.getSample();

}

}

class Sam {

}

interface Inter {

public Sam getSample();

}