Java 8 features

Lambda expressions and Functional interface

Why are Lambda expressions introduced ?

Lambda Expressions are added in Java 8 / 1.8.
We already knew java is object oriented programming language and which supports all object oriented features.
So Initially java didn’t support functional programming in java whereas python, ruby other supports.
To implement functional programming in java we are using lambda expressions.
The main object of lambda expression is introducing functional programming features in java.
Normally in object oriented programming language the data will be stored and maintained in classes and objects.
But in functional programming the data will be stored and maintained in functions and variables.

Here are some key benefits of lambda expressions:

1. **Functional Programming**: Lambdas support functional programming concepts, such as passing functions as arguments, returning them from other functions, and using higher-order functions. This makes it easier to work with operations on data in a more declarative manner.
2. **Improved Code Optimization**: By using lambdas, you can leverage the Stream API and other functional constructs to write more efficient and optimized code. For instance, using streams for processing collections can lead to better performance and cleaner code compared to traditional loops and conditional statements.
3. **Conciseness**: Lambda expressions allow you to write more compact code compared to traditional anonymous classes. They eliminate boilerplate code and make it easier to understand the purpose of a piece of code.
4. **Readability**: By reducing the verbosity, lambda expressions can make your code more readable and easier to maintain. They focus on what the code is doing rather than how it's implemented.

What is the lambda expression?

lambda expression is an **anonymous function(nameless function).** Anonymous function is **a function** which doesn’t have a **function name , access modifier and return type.**

Return type of all lambda expressions is functional interface.

The implementation of the Functional interface is lambda expression.

Java Lambda Expression Syntax

```
        (argument-list) -> {body}
```

Java lambda expression consists of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

No Parameter Syntax

```
        () -> {
        //Body of no parameter lambda
        }
```

Return Type: Lambda expressions don't explicitly state their return type. Instead, Java figures out the return type based on the method defined in the functional interface the lambda is used with.

Without Lambda Expression	**With Lambda Expression**

Case i: Lambda Expression with Zero parameter
public void normalMethod() {
System.out.println("Before Lambda Expression");
}
	()->{
	System.out.println(" Lambda Expression");
	};
	Or
	if it has a single line of statement inside the body then no need to mention {} also.
	()->System.out.println(" Lambda Expression");

Case ii: Lambda Expression with Single parameter
public void normalMethod(int a) {
System.out.println("Before Lambda Expression" + a);
}
	(int a)->{
	System.out.println("One parameter: " + a);
	};
	In lambda expression,there is no need to mention data type for variables/arguments also, as the java compiler will automatically find out the type of variable based on the context at runtime.
	(a) -> System.out.println("One parameter: " + a);

Case iii:lambda Expression with Multiple parameters
public void add(int a , int b){
System.out.println("sum:"+(a+b));
}
	(int a, int b)->{
	System.out.println("sum:"+(a+b));
	}
	if it has a single line of statement inside the body then no need to mention {} also.
	(int a , int b)->{
	System.out.println("sum:"+(a+b));
	};
	In lambda expression,there is no need to mention data type for variables/arguments also, as the java compiler will automatically

	find out the type of variable based on the context at runtime. (a,b)->System.out.println("sum:"+(a+b));

Case iv) return some value
Public void square(int a){
return (a*a);
}
	If the return keyword is used , we should declare that within{}.
	(int a)->{
	return (a*a);
	};
	Or
	If you don't want to use the return keyword then write only expression, without return and {}(remove return along with {})
	(a)->a*a;
	Or
	If it is a single parameter , no need to mention() also for input.
	a->a*a;

Functional interface

A functional interface is an interface that has **only one abstract method** and **any number of static and default methods**. These interfaces are used primarily to enable the **use of lambda expressions and method references in Java**, which were introduced in Java 8.

Implementation of any function interface is lambda expression.

Before Java 7 , we could store only abstract methods in interfaces.

In java 8, we were able to store both **default and static methods** in java.

```
@FunctionalInterface

public interface MyFunctionalInterface {

    // Single abstract method

    void abstractMethod();

    // Default method (optional)

    default void defaultMethod() {

        System.out.println("This is a default method.");

    }

    // Static method (optional)

    static void staticMethod() {

        System.out.println("This is a static method.");

    }

}
```

Key Characteristics of Functional Interfaces:

1. **Single Abstract Method (SAM)**: A functional interface must have exactly one abstract method. This method defines the target type for lambda expressions or method references.
2. **Default and Static Methods**: Functional interfaces can have any number of default or static methods. These methods do not count towards the one abstract method requirement.
3. **@FunctionalInterface Annotation**: Although not required, it is good practice to annotate functional interfaces with @FunctionalInterface. This annotation helps to ensure that the interface meets the criteria of a functional interface by generating a compile-time error if more than one abstract method is present.

Built-in functional interfaces in Java/Default functional interface in java

i) Runnable ->This interface only contains the run() method.

- **Purpose**: Represents a task that can be executed by a thread. It encapsulates a unit of work that does not return a result and cannot throw checked exceptions.
- **Abstract Method**: void run()
- **Usage**: Commonly used for defining tasks that can be executed concurrently, often with threads or thread pools.

Example:

```
Runnable runnableTask = () -> {

    System.out.println("Task is running");

};

Thread thread = new Thread(runnableTask);

thread.start();
```

ii) Comparable –> This interface only contains the compareTo() method.

Method: compareTo(T o)

Purpose: Used to define a natural ordering for objects of a class. The compareTo() method compares the current object with the specified object and returns an integer based on the ordering.

iii) Callable –> This interface only contains the call() method.

Method: call()

Purpose: Similar to Runnable, but it can return a result and throw a checked exception. The call() method is intended for tasks that need to return a result and potentially throw an exception.

iv) ActionListener –> This interface only contains the actionPerformed() method.

Method: actionPerformed(ActionEvent e)

Purpose: Used in the context of event handling in GUI applications. The actionPerformed() method is called when an action event occurs, such as a button press

Programms

Without Lambda Expression	**With Lambda Expression**

`package com.lamdaexpression;`
`@FunctionalInterface`
`interface Cab {`
`        public void bookCab();`
`}`
`class Ola implements Cab {`
`        @Override`
`        public void bookCab() {`
`                System.out.println("Ola Cab booked");`
`        }`
`}`
`public class Test {`
`        public static void main(String[] args) {`
`                Ola ol = new Ola();`
`                ol.bookCab();`
`                // Ola is a class which is implementing interface so we can write`
`                // "parent_interface variable= new child_implementation_class"`
`                Cab oll = new Ola();`
`                oll.bookCab();`
	`package com.lamdaexpression;`
	`@FunctionalInterface`
	`interface Cab {`
	`        public void bookCab();`
	`}`
	`public class Test {`
	`        public static void main(String[] args) {`
	
	`                Cab ol = ()->{System.out.println("Ola cab is booked");};`
	`                ol.bookCab();`
	`        }`
	`}`
	Output
	
	Ola cab is booked

```
	}
}
```

Output

```
Ola Cab booked
Ola Cab booked
```

Ex2: with Multiple parameters

```
@FunctionalInterface

interface Cab {
	public void bookCab(String source, String destination);
}
class Ola implements Cab {
	@Override
	public void bookCab(String source, String destination) {
		System.out.println("Ola cab is booked from " + source + " to " + destination);
	}
}
public class Test {
	public static void main(String[] args) {
		Cab c = new Ola();
		c.bookCab("wgl", "bnglr");
	}
}
```

```
@FunctionalInterface
interface Cab {
	public void bookCab(String source, String destination);
}
public class Test {
	public static void main(String[] args) {
		// Cab b = (String source, String destination) -> {
		// System.out.println("Ola cab is booked from " + source + " to " +
		// destination);
		/// };
Cab b = (source, destination) -> System.out.println("Ola cab is booked from " + source + " to " + destination);
		b.bookCab("wgl", "hyd");
	}
}
```

Output

```
Ola cab is booked from wgl to hyd
```

Output

```
Ola cab is booked from wgl to bnglr
```

```java
//with Multiple parameters having return value (without lambda expression)
@FunctionalInterface
interface Cab {
	public String bookCab(String source, String destination);
}
class Ola implements Cab {
	@Override
	public String bookCab(String source, String destination) {
		System.out.println("Ola cab is booked from " + source + " to " + destination);
		return ("price: 500");
	}
}
public class Test {
	public static void main(String[] args) {
		Cab c = new Ola();
		String str =c.bookCab("hyd", "wgl");
		System.out.println(str);
	}
}
```

Output

```java
//with Multiple parameters having return value (with lambda expression)
@FunctionalInterface
interface Cab {
	public String bookCab(String source, String destination);
}
public class Test {
	public static void main(String[] args) {
		Cab c = (String source, String destination) -> {
			System.out.println("Ola cab is booked from " + source + " to " + destination);
			return ("price: 500");
		};
		String str = c.bookCab("hyd", "wgl");
		System.out.println(str);
	}
}
```

Output

```
Ola cab is booked from hyd to wgl
price: 500
```

Ola cab is booked from hyd to wgl price: 500	

Predefined functional interfaces

Functional interface which is placed in **java.util.function** package.

- **T**: denotes the type of the input argument
- **R**: denotes the return type of the function

1. Predicate<T>

Predicate is a functional interface that represents a single argument function that returns a boolean value.(It takes single arguments as input and returns a boolean value).

It has only a Single Abstract Method test(T t), which accepts an argument of type T and returns a boolean value(true or false).

```
@FunctionalInterface

public interface Predicate<T> {

   boolean test(T t); // Abstract by default , no need to mention explicitly.

}
```

Wherever an object needs to be evaluated and a boolean value needs to be returned(or a boolean-valued Predicate exists - *in mathematical terms*) the Predicate functional interface can be used.

Ex1

```
package com.predicatefi;
import java.util.function.Predicate;
/*
* public interface Predicate<T> { boolean test(T t);  // Abstract by default , no need to mention explicitly.
* }
*/
//predicate - It takes single arguments as input and returns a boolean value.
//use only if you have conditional checks in the program.
public class Demo1 {
        public static void main(String[] args) {
                // Example 1
                Predicate<Integer> p = i -> (i < 90);
                // here Predicate<T> is an predefined functional interface and which is
                // implementing lambda expression.
                System.out.println(p.test(6)); // invoking lambda expression using test method.
                System.out.println(p.test(100)); // false
                // boolean b =p.test(6);
                // System.out.println(b);
                // Example 2
                // Check the length of given string is greater than 4 or not
                Predicate<String> st = s -> (s.length() > 4);
                System.out.println(st.test("Ramesh")); // true
                System.out.println(st.test("Ram")); // false

                //Ex 3
                //print array elements whose size is greater than 4 from array
                Predicate<String> str = s -> (s.length() > 4);
                String names[]= {"Ramesh","MSDhoni","virat","sky"};
                for (String name :names)
        {
            if(str.test(name)) {
                System.out.println(name);
            }
        }
    }
}
```

Output

```
true
false
true
false
Ramesh
MSDhoni
virat
```

Ex2

```
package com.predicatefi;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
class Employee {
    String name;
    int salary;
    int experience;
//The constructor is used to create and initialize Employee objects with specific values for name, salary, and experience. When you create a new instance of the Employee class, you pass these values as arguments, which the constructor uses to set the properties of the object.
    // The constructor is called when you instantiate an object of the class. For
    // example:
    public Employee(String name, int salary, int experience) {
        // super();
        this.name = name;
        this.salary = salary;
        this.experience = experience;
    }
    // the toString method of each Employee object will be called, and the output
    // will be a nicely formatted list of employees.
    @Override
    public String toString() {
        return "Employee{name='" + name + "', salary=" + salary + ", experience=" + experience + "}";
    }
}
public class Demo2 {
    public static void main(String[] args) {
        // EX1
        Employee em = new Employee("rohit", 34000, 9);
        // emp object--.return if salary>250000 experience>5
        Predicate<Employee> pr = e -> (e.salary > 25000 && e.experience > 5);
        System.out.println(pr.test(em)); // true
        List<Employee> emp = new ArrayList<Employee>();
        emp.add(new Employee("dhoni", 40000, 12));
        emp.add(new Employee("virat", 35000, 10));
        emp.add(new Employee("rohit", 34000, 9));
        emp.add(new Employee("kl", 11000, 5));
        emp.add(new Employee("rutuu", 10000, 4));
        emp.add(new Employee("ram", 12000, 2));
        System.out.println("list employees :" + emp);
        // emp object--.return if salary>20000 experience>5
        Predicate<Employee> p = ep -> (ep.salary > 20000 && ep.experience > 5);
        for (Employee ee : emp) {
            if (p.test(ee)) {
                System.out.println("name:" + ee.name + " " + "salary:" + ee.salary);
            }
        }
    }
}
```

Output

```
true
list employees :[Employee{name='dhoni', salary=40000, experience=12},
Employee{name='virat', salary=35000, experience=10}, Employee{name='rohit', salary=34000,
experience=9}, Employee{name='kl', salary=11000, experience=5}, Employee{name='rutuu',
salary=10000, experience=4}, Employee{name='ram', salary=12000, experience=2}]
name:dhoni salary:40000
name:virat salary:35000
name:rohit salary:34000
```

Ex3: Predicate chaining in Java allows you to combine multiple Predicate<T>.

This can be done using the and, or, and negate methods provided by the Predicate interface**.**

Joining Predicates - and , or , negate

and : The result is true only if both operands are true.

or : The result is true if at least one of the operands is true.

Not(negate) : The negate() method negates a predicate(Flips the truth value.)

```java
package com.predicatefi;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
//Joining Predicates - and  , or , negate
// and : The result is true only if both operands are true.
// or :  The result is true if at least one of the operands is true.
// Not(negate) : The negate() method negates a predicate(Flips the truth value.)
public class Demo3 {
        public static void main(String[] args) {
                // two predicate conditions
                // 1.predicate1 - checks number is even
                // 2.predicate2- checks number greater than 20
                List<Integer> l = new ArrayList<Integer>();
                l.add(5);
                l.add(10);
                l.add(15);
                l.add(20);
                l.add(25);
                l.add(30);
                l.add(35);
                l.add(40);
                l.add(45);
                // int a[]={5, 10, 15, 20, 25, 30, 35, 40, 45}; we can take array also as a
                // example.
                System.out.println("list of elements: " + l);
                Predicate<Integer> p1 = x -> x % 2 == 0;
                Predicate<Integer> p2 = y -> y > 20;
                // and
                System.out.println("using (and) operations on two predicate functional interface ");
                for (Integer i : l) {
                    if (p1.and(p2).test(i)) { // if (p1.test(i) && p2.test(i)) also get same result
                        System.out.println(i);
                    }
                }
                // or
                System.out.println(" using (or) operations on two predicate functional interface ");
                for (Integer j : l) {
                    if (p1.or(p2).test(j)) { // if(p1.test(j) || p2.test(j)) also get same result
                        System.out.println(j);
                    }
                }
                // negate
                System.out.println("predicate functional interface using (negate) operation");
                for (Integer k : l) {
                    if (p1.negate().test(k)) {
                        System.out.println(k); // It is printing Odd values as it is fliping/executing the false condition.
                    }
                }
        }
}
```

Output

```
list of elements: [5, 10, 15, 20, 25, 30, 35, 40, 45]
using (and) operations on two predicate functional interface
30
```

```
40
using (or) operations on two predicate functional interface
10
20
25
30
35
40
45
predicate functional interface using (negate) operation
5
15
25
35
45
```

2.Function<T, R>:

The **Function<T, R>** functional interface in Java is part of the **java.util.function** package and represents **a function that takes one argument of type T** and **produces a result of type R.**

Key Features:

- Single Abstract Method: It has a single abstract method called apply(T t), which takes an argument of type T and returns a value of type R

  In all scenarios where an object of a particular type is the input, an operation is performed on it and an object of another type is returned as output, the in-built functional interface Function<T, R> can be used without the need to define a new functional interface every time.

Ex1

```
package com.functionfi;
import java.util.function.Function;
public class Demo1 {
    public static void main(String[] args) {
        Function<Integer, Integer> f = n -> n * n;
        System.out.println(f.apply(8));
        System.out.println(f.apply(2));
        System.out.println(f.apply(5));
        /*
         * Integer i = f.apply(9); System.out.println(i);
         */
        // Ex2 input as string and finding string length as a output
        Function<String, Integer> f1 = s -> s.length();
        System.out.println(f1.apply("ms"));
        System.out.println(f1.apply("dhoni"));
        // Ex3
        // Function that converts Celsius to Fahrenheit
        Function<Double, Double> celsiusToFahrenheit = celsius -> (celsius * 9 / 5) +
32;
        // Applying the function
        Double fahrenheit = celsiusToFahrenheit.apply(25.0);
        System.out.println("25°C in Fahrenheit: " + fahrenheit);
        // EX4
        // Function that converts a String to uppercase
        Function<String, String> f2 = str -> str.toUpperCase();
        // Applying the function
        String input = "hello, world!";
        String uppercase = f2.apply(input);

        System.out.println("Original: " + input);
        System.out.println("Uppercase: " + uppercase);
    }
}
```

Output

```
5
25°C in Fahrenheit: 77.0
Original: hello, world!
Uppercase: HELLO, WORLD!
```

Ex2:

```
package com.functionfi;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;
class Employee {
    String ename;
    int salary;
    public Employee(String ename, int salary) {
        super();
        this.ename = ename;
        this.salary = salary;
    }
}
public class Demo2 {
    public static void main(String[] args) {
        // use case: we have list of employees in company and provided salary details.
        // we are adding bonus to them based on their salary..
        List<Employee> emp = new ArrayList<Employee>();
        emp.add(new Employee("Dhoni", 25000));
        emp.add(new Employee("kl", 35000));
        emp.add(new Employee("virat", 15000));
        emp.add(new Employee("rutuu", 45000));
        emp.add(new Employee("jaddu", 55000));
        Function<Employee, Integer> f = e -> {
            int sal = e.salary;
            if (sal >= 10000 && sal <= 20000)
                return (sal * 10 / 100);
            else if (sal > 20000 && sal <= 30000)
                return (sal * 20 / 100);
            else if (sal > 30000 && sal <= 50000)
                return (sal * 30 / 100);
            else
                return (sal * 40 / 100);
        };
        // using a predicate functional interface to check conditions . bonus>5000
        Predicate<Integer> p = b -> b > 5000;
        for (Employee em : emp) {
            // System.out.println(f.apply(em));
            int bonus = f.apply(em);
            // System.out.println(em.ename + " " + "salary=" + em.salary + " " +
"bonus:" +
            // bonus);
            // we are printing values if the conditions it true
            if (p.test(bonus)) { // calling predicate using test method
                System.out.println(em.ename + " " + "salary=" + em.salary + " " +
"bonus:" + bonus);
            }
        }
    }
}
```

Output

```
kl salary=35000 bonus:10500
rutuu salary=45000 bonus:13500
jaddu salary=55000 bonus:22000
```

Ex3:

Function chaining in Java allows you to combine multiple Function<T, R>

You can use the andThen() and compose() methods provided by the Function interface for this purpose.

andThen():The andThen() method allows you to execute the current function first and then apply another function to the result.

compose(): The compose() method allows you to apply another function first, and then apply the current function to the result

```
package com.functionfi;
import java.util.function.Function;
//Function chaining in Java allows you to combine multiple Function<T, R>
//You can use the andThen() and compose() methods provided by the Function interface for this
purpose.
public class Demo3 {
    public static void main(String[] args) {
        Function<Integer, Integer> f1 = a -> a * 2;
        Function<Integer, Integer> f2 = a -> a * a * a;
        // andThen() : first it will execute f1 reference function and that result will
        // pass to f2 reference function then print result
        System.out.println(f1.andThen(f2).apply(2));
        // compose(): first it will execute f2 reference function and that result will
        // pass to f1 reference function then print result
        System.out.println(f1.compose(f2).apply(2));
    }
}
```

Output

```
64
16
```

3.Consumer<T>

Consumer<T> **is an in-built functional interface introduced in Java 8 in the** java.util.function **package.**

It represents an operation that **accepts a single argument and returns no result**.

is often used in scenarios where you want to perform an action on a given input, such as processing or modifying it without returning any value.

The Consumer<T> interface is a SAM interface because it only has the **accept(T t)** method.

Ex1

```
package com.consumerfi;
//It represents an operation that accepts a single argument and returns no result.
import java.util.function.Consumer;
public class Demo1 {
    public static void main(String[] args) {
        // ex1
        Consumer<String> s = str -> System.out.println(str);
        s.accept("Dhoni");
        // ex2
        Consumer<Integer> s1 = st -> System.out.println(st * st);
```

```
		s1.accept(2);
		// ex3
		Consumer<String> s2 = string -> {
			System.out.println(string.toUpperCase());
			System.out.println(string.toLowerCase());
			System.out.println(string.length());
		};
		s2.accept("Virat Kohli");
		// ex4
		Consumer<String> logMessage = message -> System.out.println("Log: " +
message);
		logMessage.accept("This is a log message.");
	}
}
```

Output

```
Dhoni
4
VIRAT KOHLI
virat kohli
11
Log: This is a log message.
```

Ex2

```
package com.consumerfi;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
```

```java
import java.util.function.Function;
import java.util.function.Predicate;
class Employee {
	String ename;
	int salary;
	String gender;
	public Employee(String ename, int salary, String gender) {
		super();
		this.ename = ename;
		this.salary = salary;
		this.gender = gender;
	}
}
public class Demo2 {
	public static void main(String[] args) {
		// use case: we have list of employees in company and provided salary details.
		// we are adding bonus to them based on their salary and returning..
		// we are printing the values.
		List<Employee> emp = new ArrayList<Employee>();
		emp.add(new Employee("Dhoni", 25000, "Male"));
		emp.add(new Employee("smriti", 35000, "Female"));
		emp.add(new Employee("virat", 15000, "Male"));
		emp.add(new Employee("richa", 65000, "Female"));
		emp.add(new Employee("jaddu", 55000, "Male"));
		// task 1: Function functional interface (we are adding bonus to them based on
		// their
		// salary and returning..)
		Function<Employee, Integer> e = a -> (a.salary * 10) / 100;
		// task2: Predicate Functional interface
		Predicate<Integer> p = b -> b >= 5000;
```

```
                // task 3: Consumer Functional interface (we are just printing the values.)
                Consumer<Employee> co = c -> System.out.println(c.ename + " " + c.salary + " " + c.gender);
                for (Employee ee : emp) {
                        int bonus = e.apply(ee); // invoked function
                        if (p.test(bonus)) { // invoked predicate
                                co.accept(ee); // invoked consumer
                                System.out.println("Employee Bonus:" + bonus);
                        }
                }
        }
}
```

Output

```
richa 65000 Female
Employee Bonus:6500
jaddu 55000 Male
Employee Bonus:5500
```

Ex:3

Chaining:

andThen(): The andThen method is used to combine the two consumers into a single consumer. When you call accept on this combined consumer, it executes the first consumer, then the second one.

```
package com.consumerfi;
import java.util.function.Consumer;
```

```
//consumer chaining
//andThen: The andThen method is used to combine the two consumers into a single consumer. When you call accept on this combined consumer, it executes the first consumer, then the second one.
public class Demo3 {
    public static void main(String[] args) {
        Consumer<String> c1 = str -> System.out.println(str + " " + "is white");
        Consumer<String> c2 = str -> System.out.println(str + " " + "is having four legs");
        Consumer<String> c3 = str -> System.out.println(str + " " + "is eating grass");
        /*
         * c1.accept("Cow"); c2.accept("Cow"); c3.accept("Cow");
         */
        // or
        /*
         * Consumer<String> c4 = c1.andThen(c2).andThen(c3); c4.accept("Cow");
         */
        // or
        c1.andThen(c2).andThen(c3).accept("Cow");
    }
}
```

Output

```
Cow is white
Cow is having four legs
Cow is eating grass
```

4. Supplier<R>

It has a single abstract method, get(), **which does not take any arguments but returns a value of type R**.

`() -> R`

Key Features of Supplier

1. **No Input Arguments**: Unlike Function or Consumer, a Supplier does not accept any input.
2. **Return Type**: It returns a value of the specified type R.

```
package com.supplierfi;
import java.util.Date;
import java.util.function.Supplier;
public class Demo1 {
        public static void main(String[] args) {
                // EX1
                Supplier<Date> s = () -> new Date();
                System.out.println(s.get());
                // ex2
                Supplier<String> stringSupplier = () -> "Hello, Supplier!";
                // Using the Supplier
                System.out.println(stringSupplier.get()); // Output: Hello, Supplier!
                // EX3
                // Create a Supplier that supplies a random number
                Supplier<Double> supl = () -> Math.random();
                // Get a random number
                Double randomNumber = supl.get();
                System.out.println("Random Number: " + randomNumber);
                // ex4
                Supplier<Long> currentTimeSupplier = () -> System.currentTimeMillis();
                System.out.println(currentTimeSupplier.get());

                //EX5
                Supplier<Integer> ss = () -> Math.min(12, 13);
                System.out.println(ss.get());
        }
}
```

Output

```
Hello, Supplier!
Random Number: 0.7025954146371134
1727085264208
12
```

Ex2:

```
package com.supplierfi;
import java.util.function.Supplier;
//Manual Chaining
public class Demo2 {
        public static void main(String[] args) {
                Supplier<String> supplier1 = () -> "Hello";
                Supplier<String> supplier2 = () -> "World";
                // Manually chain the suppliers
                Supplier<String> combinedSupplier = () -> supplier1.get() + " " + supplier2.get();
                // Output the combined result
                System.out.println(combinedSupplier.get()); // Output: Hello World
        }
}
```

Output

```
Hello World
```

Functional Interface	**Respective SAM**	**Chaining methods**(we use //Manual Chaining For every functional interface)

Predicate<T> **Predicate** is a functional interface that represents a single argument function that returns a boolean value.(It takes single arguments as input and returns a boolean value).	It has only a Single Abstract Method **test(T t),** which accepts an argument of type T and returns a boolean value(true or false)	Predicate chaining in Java allows you to combine multiple Predicate<T>. Joining Predicates - **and , or , negate** **and :** The result is true only if both operands are true. **or :** The result is true if at least one of the operands is true. **Not(negate) :** The negate() method negates a predicate(Flips the truth value.) **//Manual Chaining**
Function<T, R>: Represents a function that takes one argument of type T and produces a result of type R**.**	It has a single abstract method called **apply(T t),** which takes an argument of type T and returns a value of type R	Function chaining in Java allows you to combine multiple Function<T, R>. **andThen():**The andThen() method allows you to execute the current function **first** and then apply another function to the result. **compose():** The compose() method allows you to apply another function first, and then apply the current function to the result **//Manual Chaining**
Consumer<T> It represents an operation that	The Consumer<T> interface is a SAM interface because it	**andThen():** The andThen method is used to combine the two consumers into a single consumer. When you call

accepts a single argument and returns no result.	only has the **accept(T t)** method.	accept on this combined consumer, it executes the first consumer, then the second one. **//Manual Chaining**

Supplier<R>
which does not take any arguments but returns a value of type R.
	It has a single abstract method, **get()**,	**//Manual Chaining**
		Supplier<String> supplier1 = () -> "Hello";
		Supplier<String> supplier2 = () -> "World";
		// Manually chain the suppliers
		Supplier<String> combinedSupplier = () -> supplier1.get() + " " +supplier2.get();
		// Output the combined result
		System.*out*.println(combinedSupplier.get());
		// Output: Hello World