FUNCTIONAL INTERFACE

What is a Functional Interface?

A Functional Interface is an interface that has only one abstract method. It can have multiple default or static methods, but only one abstract method.

Functional Interfaces are the backbone of Lambda Expressions in Java 8.

How to Define a Functional Interface

```
@FunctionalInterface
    interface MyFunctionalInterface {
        void sayHello(); // Only one abstract method
    }
```

The @FunctionalInterface annotation is **optional**, but it helps the compiler ensure the interface has only one abstract method.

🔷 **Built-in Functional Interfaces in Java 8**

Java 8 provides many ready-to-use functional interfaces in the java.util.function package. Let's go through the most important ones with **simple examples**.

Interface	Input	Output	Use Case
Predicate<T>	T	boolean	Condition check
Function<T,R>	T	R	Transform data
Consumer<T>	T	void	Perform action
Supplier<T>	None	T	Supply value
BiFunction<T,U,R>	T, U	R	Combine two inputs

1. Predicate<T>

🔶 What is it?

- Predicate<T> is a **functional interface** in Java 8.
- It represents a **boolean-valued function** of one argument.
- It is often used for **filtering or checking conditions**.

🔶 Method in Predicate

boolean test(T t);

- **This method takes an input of type T and returns true or false.**

- **Example Program**

```
import java.util.function.Predicate;
public class PredicateExample {
        public static void main(String[] args) {
                // Step 1: Create a Predicate to check if a string length is greater than 5
               Predicate isLong = str -> str.length() > 5;
                // Step 2: Test the predicate with different strings
                System.out.println(isLong.test("Hello"));     // false
               System.out.println(isLong.test("Functional")); // true
        }
}
```

Explanation

1. **Predicate<String>** means we are working with a string input.
2. **str -> str.length() > 5** is a **lambda expression** that checks if the string's length is more than 5.
3. **isLong.test("Hello")** runs the check:
    - "Hello" has 5 characters → false
4. **isLong.test("Functional")**:
    - "Functional" has 10 characters → true

Why do we use isLong.test(...) inside System.out.println(...)?

🔶 1. What is isLong?

```
Predicate isLong = str -> str.length() > 5;
```

- This line creates a **Predicate** that checks if a string has more than 5 characters.
- isLong is now a **function** that takes a string and returns true or false.

🔶 2. What is .test(...)?

- .test(...) is the **method** of the Predicate interface.
- It is used to **run the condition** defined in the lambda expression.

So when you write:

```
isLong.test("Hello")
```

It checks:
➡️ "Hello".length() > 5
➡️ 5 > 5 → false

🔶 3. Why inside System.out.println(...)?

Because we want to **print the result** of the test to the console.

```
System.out.println(isLong.test("Hello"));
```

This prints false because "Hello" is not longer than 5 characters.

🔁 Summary

Code	What it does
isLong	A function that checks if a string is long
isLong.test("Hello")	Runs the check and returns false
System.out.println(...)	Prints the result to the screen

Step-by-Step: Check Even Numbers Using Predicate

🔶 1. Theory

- We use Predicate<Integer> because we are working with **integers**.
- The .test() method will check if the number is divisible by 2.
- If number % 2 == 0, then the number is even.

---

2. Code Example

```
import java.util.function.Predicate;
        public class EvenNumberCheck {
                public static void main(String[] args) {
                 // Step 1: Create a Predicate to check if a number is even
                 Predicate<Integer>  isEven = num -> num % 2 == 0;
                 // Step 2: Test the predicate with different numbers
                 System.out.println("Is 4 even? " + isEven.test(4)); // true
                System.out.println("Is 7 even? " + isEven.test(7)); // false
                 System.out.println("Is 10 even? " + isEven.test(10)); // true
        }
}
```

Explanation

1. **Predicate<Integer> isEven**: This defines a function that takes an integer and returns a boolean.
2. **num -> num % 2 == 0**: This lambda checks if the number is divisible by 2.
3. **.test(4)**:
    - 4 % 2 == 0 → true
4. **.test(7)**:
    - 7 % 2 == 1 → false
5. **System.out.println(...)**: Prints the result of each check.

❖ So now my question is how did predicate became a method of functional interface . how i know it have only one abstract method

🔷 1. What Makes an Interface a Functional Interface?

A **functional interface** in Java is:

- An interface that has **exactly one abstract method**.
- It can have **default** or **static** methods, but only **one abstract method**.
- It can be annotated with @FunctionalInterface (optional but recommended).

🔷 2. How is Predicate a Functional Interface?

The Predicate<T> interface is part of the **java.util.function** package in Java 8.

Here's a simplified version of how it looks:

```
@FunctionalInterface
public interface Predicate {
        boolean test(T t); // Only one abstract method
        // These are default methods (not abstract)
        default Predicate and(Predicate other) { ... }
        default Predicate or(Predicate other) { ... }
        default Predicate negate() { ... }
        // Static method
        static  Predicate isEqual(Object targetRef) { ... }
    }
```

So why is it a functional interface?

Because it has **only one abstract method**:

```
boolean test(T t);
```

All the other methods (and, or, negate, isEqual) are either **default** or **static**, which do **not** count as abstract methods.

3. How Can You Know This Yourself?

You can check in **two ways**:

Option 1: Use the Java Docs

Go to the official Java documentation and look at the method summary. You'll see that only test(T t) is abstract.

Option 2: Use Your IDE (like IntelliJ or Eclipse)

- Hover over Predicate or press **Ctrl + Click** on it.
- It will take you to the source code.
- You'll see the @FunctionalInterface annotation and the test() method.

🔍 **Summary**

Concept	Explanation
Predicate<T>	A built-in functional interface in Java 8
Abstract Method	boolean test(T t)
Why Functional?	Only one abstract method, others are default/static
How to Check?	Java Docs or your IDE

2. Function<T, R>

🔶 **What is it?**

- Function<T, R> is a **functional interface** that takes an input of type T and returns a result of type R.
- It is used for **data transformation** — converting one type to another or modifying data.

🔶 **Abstract Method**

```
R apply(T t);
```

- Takes one input (T) and returns one output (R).

Real-Life Analogy

Think of it like a **machine**:

- You give it a raw material (input T)

- It processes it and gives you a finished product (output R)

Example: Convert a String to its Length

```
import java.util.function.Function;
public class FunctionExample {
public static void main(String[] args) {
    // Step 1: Create a Function that takes a String and returns its length
    Function<String, Integer> getLength = str -> str.length();
    // Step 2: Apply the function to different strings
    System.out.println("Length of 'Java': " + getLength.apply("Java"));      // 4
System.out.println("Length of 'Functional': " + getLength.apply("Functional")); // 10
}
}
```

🔍 **Explanation**

1. **Function<String, Integer>**:
    - Input: String
    - Output: Integer (length of the string)
2. **Lambda: str -> str.length()**:
    - Takes a string and returns its length.
3. **getLength.apply("Java")**:
    - "Java".length() → 4
4. **System.out.println(...)**:
    - Prints the result of the function.

5. 🔁 Summary

Concept	Meaning
`Function<T, R>`	Takes input `T`, returns output `R`
Abstract Method	`R apply(T t)`

Concept	Meaning
Use Case	Data transformation (e.g., String → Integer)

3. Consumer<T>

🔶 **What is it?**

- Consumer<T> is a **functional interface** that takes an input of type T and **performs an action** on it.
- It **does not return** any result (i.e., return type is void).

🔶 **Abstract Method**

```
void accept(T t);
```

- Takes one input and performs some operation (like printing, saving, modifying, etc.)

Real-Life Analogy

Imagine a **printer**:

- You give it a document (input T)
- It prints it (performs an action)
- But it doesn't give anything back (no return value)

Example: Print a Name

```
import java.util.function.Consumer;
public class ConsumerExample {
   public static void main(String[] args) {
      // Step 1: Create a Consumer that prints a name
      Consumer printName = name -> System.out.println("Hello, " + name + "!");
      // Step 2: Use the consumer
      printName.accept("John");   // Hello, John!
      printName.accept("Alice");  // Hello, Alice!
```

```
    }
}
```

🔍 **Explanation**

1. **Consumer<String>**: This means we are working with a string input.
2. **Lambda: name -> System.out.println(...)**: This prints a greeting.
3. **printName.accept("John")**:
    - Calls the accept() method with "John" as input.
    - Prints: Hello, John!

🔁 **Summary**

Concept	Meaning
Consumer<T>	Takes input T, performs an action
Abstract Method	void accept(T t)
Use Case	Logging, printing, saving, etc.

4. Supplier<T>

🔶 **What is it?**

- Supplier<T> is a **functional interface** that **does not take any input** but **returns a value** of type T.
- It's used when you want to **generate or supply values**, like random numbers, current time, or default values.

🔶 **Abstract Method**

```
T get();
```

- No parameters.
- Returns a value of type T.

Real-Life Analogy

Imagine a **vending machine**:

- You press a button (no input)
- It gives you a snack (output)

Example 1: Supply a random number

```
import java.util.function.Supplier;
public class RandomNumberSupplier {
   public static void main(String[] args) {
      Supplier<Double> randomSupplier = () -> Math.random();
      System.out.println("Random number: " + randomSupplier.get());
      System.out.println("Another random number: " + randomSupplier.get());
   }
}
```

🔍 **Explanation**

1. **Supplier<Double>**: This means it will return a Double value.
2. **Lambda: () -> Math.random()**: No input, just returns a random number.
3. **randomSupplier.get()**: Calls the supplier to get a value.

Example 2: Supply current date and time

```
import java.util.function.Supplier;
import java.time.LocalDateTime;
public class DateTimeSupplier {
   public static void main(String[] args) {
      Supplier dateTimeSupplier = () -> LocalDateTime.now();
      System.out.println("Current time: " + dateTimeSupplier.get());
   }
}
```

Example 3: Supply a default message

```
import java.util.function.Supplier;
public class DefaultMessageSupplier {
   public static void main(String[] args) {
```

```
        Supplier messageSupplier = () -> "No data available";
        System.out.println("Message: " + messageSupplier.get());
    }
}
```

🔁 Summary

Concept	Meaning
Supplier<T>	Takes no input, returns a value of type T
Abstract Method	T get()
Use Case	Generating values, default values, lazy loading