# Functional Interface

### ****What is a Functional Interface?****

A **Functional Interface** is an interface in Java that contains exactly one abstract method. It can have multiple default or static methods, but it must have only one abstract method. Functional interfaces are central to enabling lambda expressions in Java, as they define the target type for lambda expressions or method references.

### ****Characteristics of Functional Interfaces:****

1. **Single Abstract Method (SAM):** A functional interface must have exactly one abstract method. This is what makes it compatible with lambda expressions.
2. **Can Have Default and Static Methods:** In addition to the single abstract method, a functional interface can have multiple default or static methods. These methods do not affect its status as a functional interface.
3. **@FunctionalInterface Annotation:** While not mandatory, the @FunctionalInterface annotation is often used to indicate that an interface is intended to be a functional interface. The compiler can verify this at compile time, ensuring that the interface adheres to the rules of a functional interface.

### ****Syntax of a Functional Interface****

@FunctionalInterface

public interface MyFunctionalInterface {

void myMethod(); // Single abstract method (SAM)

default void defaultMethod() {

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

}

static void staticMethod() {

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

}

}

In the above example:

* myMethod() is the abstract method.
* defaultMethod() is a default method.
* staticMethod() is a static method.

### ****Key Points About Functional Interfaces****

* **Single Abstract Method**: The functional interface must have exactly one abstract method, which represents the functionality the interface is intended to encapsulate.
* **Default Methods**: These are methods with a body, and they provide the ability to add methods to interfaces without breaking existing implementations.
* **Static Methods**: These are methods with a body that belong to the interface itself, not to any instances of the interface.

List of **functional interfaces** in Java (from java.util.function and core Java), along with their **single abstract method**:

### ****Core Functional Interfaces****

1. **Runnable**  
   void run()
2. **Callable**  
   V call() throws Exception
3. **Comparable**  
   int compareTo(T o)
4. **Comparator**  
   int compare(T o1, T o2)

### ****java.util.function Package****

#### ****Predicates****

1. **Predicate**  
   boolean test(T t)
2. **BiPredicate<T, U>**  
   boolean test(T t, U u)

#### ****Functions****

1. **Function<T, R>**  
   R apply(T t)
2. **BiFunction<T, U, R>**  
   R apply(T t, U u)
3. **UnaryOperator**  
   T apply(T t)
4. **BinaryOperator**  
   T apply(T t1, T t2)

#### ****Consumers****

1. **Consumer**  
   void accept(T t)
2. **BiConsumer<T, U>**  
   void accept(T t, U u)

#### ****Suppliers****

1. **Supplier**  
   T get()

### ****Primitive Specializations****

#### ****Int****

1. **IntPredicate**  
   boolean test(int value)
2. **IntFunction**  
   R apply(int value)
3. **IntConsumer**  
   void accept(int value)
4. **IntSupplier**  
   int getAsInt()
5. **IntUnaryOperator**  
   int applyAsInt(int operand)
6. **IntBinaryOperator**  
   int applyAsInt(int left, int right)

#### ****Long****

1. **LongPredicate**  
   boolean test(long value)
2. **LongFunction**  
   R apply(long value)
3. **LongConsumer**  
   void accept(long value)
4. **LongSupplier**  
   long getAsLong()
5. **LongUnaryOperator**  
   long applyAsLong(long operand)
6. **LongBinaryOperator**  
   long applyAsLong(long left, long right)

#### ****Double****

1. **DoublePredicate**  
   boolean test(double value)
2. **DoubleFunction**  
   R apply(double value)
3. **DoubleConsumer**  
   void accept(double value)
4. **DoubleSupplier**  
   double getAsDouble()
5. **DoubleUnaryOperator**  
   double applyAsDouble(double operand)
6. **DoubleBinaryOperator**  
   double applyAsDouble(double left, double right)

#### ****ToXXX Functions****

1. **ToIntFunction**  
   int applyAsInt(T value)
2. **ToLongFunction**  
   long applyAsLong(T value)
3. **ToDoubleFunction**  
   double applyAsDouble(T value)
4. **IntToLongFunction**  
   long applyAsLong(int value)
5. **IntToDoubleFunction**  
   double applyAsDouble(int value)
6. **LongToIntFunction**  
   int applyAsInt(long value)
7. **LongToDoubleFunction**  
   double applyAsDouble(long value)
8. **DoubleToIntFunction**  
   int applyAsInt(double value)
9. **DoubleToLongFunction**  
   long applyAsLong(double value)

## ****Common Functional Interfaces in Java 8****

Java 8 introduced several built-in functional interfaces in the java.util.function package. These interfaces are commonly used for functional programming tasks, especially when working with lambdas.

### 1. Predicate<T>

A Predicate is a functional interface that represents a boolean-valued function of one argument.

#### ****Method:****

* boolean test(T t): Evaluates this predicate on the given argument.

**Example:**

import java.util.function.Predicate;

public class PredicateExample {

public static void main(String[] args) {

Predicate<Integer> isEven = (n) -> n % 2 == 0;

System.out.println(isEven.test(4)); // true

System.out.println(isEven.test(5)); // false

}

}

### 2. Function<T, R>

The Function interface represents a function that accepts one argument and produces a result.

#### ****Method:****

* R apply(T t): Applies this function to the given argument.

**Example:**

import java.util.function.Function;

public class FunctionExample {

public static void main(String[] args) {

Function<String, Integer> stringLength = (str) -> str.length();

System.out.println(stringLength.apply("Hello")); // 5

}

}

### 3. Consumer<T>

A Consumer represents an operation that accepts a single input argument and returns no result.

#### ****Method:****

* void accept(T t): Performs the operation on the given argument.

**Example:**

import java.util.function.Consumer;

public class ConsumerExample {

public static void main(String[] args) {

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

print.accept("Hello World"); // Outputs: Hello World

}

}

### 4. Supplier<T>

A Supplier represents a supplier of results. It takes no input but returns a result.

#### ****Method:****

* T get(): Gets a result.

**Example:**

import java.util.function.Supplier;

public class SupplierExample {

public static void main(String[] args) {

Supplier<String> greet = () -> "Hello, Supplier!";

System.out.println(greet.get()); // Outputs: Hello, Supplier!

}

}

### 5. UnaryOperator<T>

A UnaryOperator is a specialized form of Function, where the input and output are of the same type.

#### ****Method:****

* T apply(T t): Applies this operator to the given argument.

**Example:**

import java.util.function.UnaryOperator;

public class UnaryOperatorExample {

public static void main(String[] args) {

UnaryOperator<Integer> square = (n) -> n \* n;

System.out.println(square.apply(5)); // Outputs: 25

}

}

### 6. BinaryOperator<T>

A BinaryOperator is a specialized form of BiFunction, where the operands and the result are all of the same type.

#### ****Method:****

* T apply(T t1, T t2): Applies this operator to the given arguments.

**Example:**

import java.util.function.BinaryOperator;

public class BinaryOperatorExample {

public static void main(String[] args) {

BinaryOperator<Integer> add = (a, b) -> a + b;

System.out.println(add.apply(3, 7)); // Outputs: 10

}

}

## ****Examples of Using Lambda Expressions with Functional Interfaces****

Lambda expressions are often used to implement functional interfaces. Here are a few examples.

### ****1. Using**** Predicate ****with Lambda Expression****

import java.util.function.Predicate;

public class LambdaWithPredicate {

public static void main(String[] args) {

Predicate<String> isNotEmpty = (str) -> !str.isEmpty();

System.out.println(isNotEmpty.test("Hello")); // true

System.out.println(isNotEmpty.test("")); // false

}

}

### ****2. Using**** Function ****with Lambda Expression****

import java.util.function.Function;

public class LambdaWithFunction {

public static void main(String[] args) {

Function<String, Integer> stringLength = (str) -> str.length();

System.out.println(stringLength.apply("Java")); // 4

System.out.println(stringLength.apply("Lambda Expressions")); // 19

}

}

### ****3. Using**** Consumer ****with Lambda Expression****

import java.util.function.Consumer;

public class LambdaWithConsumer {

public static void main(String[] args) {

Consumer<String> printUpperCase = (str) -> System.out.println(str.toUpperCase());

printUpperCase.accept("hello"); // HELLO

printUpperCase.accept("world"); // WORLD

}

}

### ****4. Using**** Supplier ****with Lambda Expression****

import java.util.function.Supplier;

public class LambdaWithSupplier {

public static void main(String[] args) {

Supplier<String> getMessage = () -> "Hello, Supplier!";

System.out.println(getMessage.get()); // Hello, Supplier!

}

}

### ****5. Using**** UnaryOperator ****with Lambda Expression****

import java.util.function.UnaryOperator;

public class LambdaWithUnaryOperator {

public static void main(String[] args) {

UnaryOperator<Integer> doubleValue = (n) -> n \* 2;

System.out.println(doubleValue.apply(5)); // 10

}

}

## ****Advantages of Functional Interfaces****

1. **Encapsulation of Behavior**: Functional interfaces allow you to encapsulate behavior, making your code more modular and flexible.
2. **Simplified Code**: Using functional interfaces with lambda expressions simplifies code, making it more concise and readable.
3. **Enables Functional Programming**: Functional interfaces are a key enabler for functional programming in Java. They allow you to pass behavior as parameters (for example, when using streams).
4. **Better Use of the Streams API**: Functional interfaces work seamlessly with Java’s Streams API, which enables operations like map, filter, reduce, etc.