 Consumer<T> is used for performing an action without returning anything.

 Supplier<T> provides values without taking input.

 Function<T, R> transforms data.

 Predicate<T> is used for conditions and filtering.

1. Consumer<T> :

Consumes Input, No Return.

A Consumer<T> is a functional interface that **accepts an input** but **does not return anything**. It is used when we want to **perform some operation on an input** without returning a value.

import java.util.function.Consumer;

public class ConsumerExample {

public static void main(String[] args) {

// Consumer that prints a string

Consumer<String> printMessage = message -> System.out.println("Message: " + message);

// Using accept() method

printMessage.accept("Hello, World!"); // Output: Message: Hello, World!

}

}

## **2.** Supplier<T> **(Produces Output, No Input)**

A Supplier<T> is a functional interface that **does not take any input** but **returns an output**. It is used when you need to **supply values** (e.g., generate numbers, fetch data).

**Example:** Generating a random number

import java.util.function.Supplier;

import java.util.Random;

public class SupplierExample {

public static void main(String[] args) {

// Supplier that generates a random number

Supplier<Integer> randomNumberSupplier = () -> new Random().nextInt(100);

// Using get() method

System.out.println("Random Number: " + randomNumberSupplier.get());

}

}

## 3. Function<T, R> **(Takes Input, Returns Output)**

A Function<T, R> is a functional interface that **takes one input (T)** and **returns a result (R)**. It is useful when you need to **transform data**.

**Example:** Converting a string to uppercase

import java.util.function.Function;

public class FunctionExample {

public static void main(String[] args) {

// Function that converts a string to uppercase

Function<String, String> toUpperCase = str -> str.toUpperCase();

// Using apply() method

System.out.println(toUpperCase.apply("hello")); // Output: HELLO

}

}

4. Predicate<T> (Takes Input, Returns Boolean)

A Predicate<T> is a functional interface that **takes an input (T)** and **returns a boolean (true/false)**. It is useful for **filtering data**.

### **Example:** Checking if a number is even

import java.util.function.Predicate;

public class PredicateExample {

public static void main(String[] args) {

// Predicate that checks if a number is even

Predicate<Integer> isEven = num -> num % 2 == 0;

// Using test() method

System.out.println(isEven.test(10)); // Output: true

System.out.println(isEven.test(7)); // Output: false

}

}

CUSTOM FUNCTIONAL INTERFACE:

**Example:** Custom functional interface for adding two numbers

@FunctionalInterface

interface MathOperation {

int operate(int a, int b);

}

public class CustomFunctionalInterfaceExample {

public static void main(String[] args) {

// Implementing the interface using a lambda expression

MathOperation addition = (a, b) -> a + b;

// Using the custom functional interface

System.out.println("Sum: " + addition.operate(5, 3)); // Output: 8

}

}

**Example:** Function + Consumer

import java.util.function.Function;

import java.util.function.Consumer;

public class FunctionConsumerExample {

public static void main(String[] args) {

// Function to calculate square

Function<Integer, Integer> squareFunction = x -> x \* x;

// Consumer to print the result

Consumer<Integer> printResult = result -> System.out.println("Result: " + result);

// Combining Function and Consumer

int number = 4;

int square = squareFunction.apply(number); // Function usage

printResult.accept(square); // Consumer usage

}

}

Method Reference in Java

A method reference is a shorthand notation of a lambda expression to call a method. It allows you to refer to a method without executing it. Method references enhance code readability and make it more concise.

**Types of Method References**

There are four types of method references in Java:

1. **Reference to a Static Method**
2. **Reference to an Instance Method of a Particular Object**
3. **Reference to an Instance Method of an Arbitrary Object of a Particular Type**
4. **Reference to a Constructor**

### **1. Reference to a Static Method**

You can refer to a static method using the syntax:

ClassName::methodName

Example

import java.util.function.Function;

public class StaticMethodReferenceExample {

public static int square(int number) {

return number \* number;

}

public static void main(String[] args) {

// Lambda Expression

Function<Integer, Integer> lambdaSquare = number -> StaticMethodReferenceExample.square(number);

// Method Reference

Function<Integer, Integer> methodRefSquare = StaticMethodReferenceExample::square;

System.out.println("Square using Lambda: " + lambdaSquare.apply(5));

System.out.println("Square using Method Reference: " + methodRefSquare.apply(5));

}

}

1. Reference to an Instance Method of a Particular Object

You can refer to an instance method of a specific object using the syntax:

instance::methodName

Example:

import java.util.function.Consumer;

public class InstanceMethodReferenceExample {

public void printMessage(String message) {

System.out.println(message);

}

public static void main(String[] args) {

InstanceMethodReferenceExample example = new InstanceMethodReferenceExample();

// Lambda Expression

Consumer<String> lambdaPrinter = message -> example.printMessage(message);

// Method Reference

Consumer<String> methodRefPrinter = example::printMessage;

lambdaPrinter.accept("Hello from Lambda!");

methodRefPrinter.accept("Hello from Method Reference!");

}

}

### **3. Reference to an Instance Method of an Arbitrary Object of a Particular Type**

You can refer to an instance method of any object of a specific type using the syntax:

ClassName::methodName

import java.util.function.Function;

import java.util.Arrays;

import java.util.List;

public class ArbitraryObjectMethodReferenceExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Lambda Expression

names.forEach(name -> System.out.println(name));

// Method Reference

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

}

}

### **4. Reference to a Constructor**

You can refer to a constructor using the syntax:

ClassName::new

import java.util.function.Function;

class Person {

private String name;

public Person(String name) {

this.name = name;

}

@Override

public String toString() {

return "Person{name='" + name + "'}";

}

}

public class ConstructorReferenceExample {

public static void main(String[] args) {

// Lambda Expression

Function<String, Person> lambdaConstructor = name -> new Person(name);

// Method Reference

Function<String, Person> constructorRef = Person::new;

Person p1 = lambdaConstructor.apply("Alice");

Person p2 = constructorRef.apply("Bob");

System.out.println(p1);

System.out.println(p2);

}

}

Example 1: Reference to a No-Argument Instance Method

import java.util.function.Supplier;

class Example {

public String greet() {

return "Hello, World!";

}

}

public class NoParameterMethodReference {

public static void main(String[] args) {

Example example = new Example();

// Lambda Expression

Supplier<String> lambdaGreet = () -> example.greet();

// Method Reference

Supplier<String> methodRefGreet = example::greet;

System.out.println("Lambda: " + lambdaGreet.get());

System.out.println("Method Reference: " + methodRefGreet.get());

}

}

### **Example 2: Reference to a Static No-Argument Method**

Static methods with no arguments can also be referenced.

import java.util.function.Supplier;

class Example {

public static String staticGreet() {

return "Hello from Static Method!";

}

}

public class StaticNoParameterMethodReference {

public static void main(String[] args) {

// Lambda Expression

Supplier<String> lambdaStaticGreet = () -> Example.staticGreet();

// Method Reference

Supplier<String> methodRefStaticGreet = Example::staticGreet;

System.out.println("Lambda: " + lambdaStaticGreet.get());

System.out.println("Method Reference: " + methodRefStaticGreet.get());

}

}