**Java Builtin Functional Interfaces Examples**

Java 8 introduced the concept of functional interfaces as part of the Java Stream API and the wider adoption of lambda expressions. These interfaces represent single abstract methods (SAMs) that can be used as the target for lambda expressions, method references, or functional programming in general. Let's dive into each of these key functional interfaces with detailed examples.

**1. Function Interface**

The Function<T, R> interface represents a function that takes one argument of type T and produces a result of type R.

**Example: Convert String to Integer**

import java.util.function.Function;

public class FunctionExample {

public static void main(String[] args) {

// Function to convert a string to its integer length

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

// Apply the function

int length = stringLength.apply("Hello, World!");

System.out.println("Length: " + length); // Output: Length: 13

}

}

**2. BiFunction Interface**

The BiFunction<T, U, R> interface represents a function that takes two arguments of types T and U, and produces a result of type R.

**Example: Add Two Integers**

import java.util.function.BiFunction;

public class BiFunctionExample {

public static void main(String[] args) {

// BiFunction to add two integers

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

// Apply the BiFunction

int sum = add.apply(5, 10);

System.out.println("Sum: " + sum); // Output: Sum: 15

}

}

**3. Predicate Interface**

The Predicate<T> interface represents a boolean-valued function that takes one argument of type T.

**Example: Check if a Number is Even**

import java.util.function.Predicate;

public class PredicateExample {

public static void main(String[] args) {

// Predicate to check if a number is even

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

// Test the predicate

boolean result = isEven.test(4);

System.out.println("Is 4 even? " + result); // Output: Is 4 even? true

}

}

**4. Supplier Interface**

The Supplier<T> interface represents a function that takes no arguments and produces a result of type T.

**Example: Supply a Default String**

import java.util.function.Supplier;

public class SupplierExample {

public static void main(String[] args) {

// Supplier to provide a default string

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

// Get the result from the supplier

String result = defaultString.get();

System.out.println(result); // Output: Hello, Supplier!

}

}

**5. Consumer Interface**

The Consumer<T> interface represents an operation that takes one argument of type T and returns no result.

**Example: Print a String**

import java.util.function.Consumer;

public class ConsumerExample {

public static void main(String[] args) {

// Consumer to print a string

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

// Accept the string to print it

print.accept("Hello, Consumer!"); // Output: Hello, Consumer!

}

}

**6. UnaryOperator Interface**

The UnaryOperator<T> interface is a special case of Function where both the argument and result are of the same type.

**Example: Square a Number**

import java.util.function.UnaryOperator;

public class UnaryOperatorExample {

public static void main(String[] args) {

// UnaryOperator to square a number

UnaryOperator<Integer> square = num -> num \* num;

// Apply the operator

int result = square.apply(5);

System.out.println("Square: " + result); // Output: Square: 25

}

}

**7. BinaryOperator Interface**

The BinaryOperator<T> interface is a special case of BiFunction where both operands and the result are of the same type.

**Example: Find Maximum of Two Numbers**

import java.util.function.BinaryOperator;

public class BinaryOperatorExample {

public static void main(String[] args) {

// BinaryOperator to find the maximum of two numbers

BinaryOperator<Integer> max = (a, b) -> a > b ? a : b;

// Apply the operator

int result = max.apply(10, 20);

System.out.println("Max: " + result); // Output: Max: 20

}

}

**Combining Functional Interfaces**

One of the powerful aspects of functional interfaces is that you can combine them to perform more complex operations. For example, you can chain functions together using the andThen or compose methods available on the Function interface.

**Example: Chain Functions**

import java.util.function.Function;

public class ChainFunctionExample {

public static void main(String[] args) {

// Function to double a number

Function<Integer, Integer> doubleIt = num -> num \* 2;

// Function to add 10

Function<Integer, Integer> addTen = num -> num + 10;

// Chain the functions: first double, then add 10

Function<Integer, Integer> doubleAndAddTen = doubleIt.andThen(addTen);

// Apply the chained function

int result = doubleAndAddTen.apply(5);

System.out.println("Result: " + result); // Output: Result: 20

}

}

**Conclusion**

These functional interfaces in Java 8 provide powerful tools for writing concise, functional-style code. They are widely used in the Java Stream API, lambdas, and method references, enabling developers to write more declarative and maintainable code.

Understanding these interfaces will help you make better use of Java's functional programming capabilities, especially when working with streams, collections, and other data processing tasks.