**Functional Interfaces in Java**

Java 8 introduced functional interfaces, which are interfaces with a single abstract method. They allow us to write concise and readable code, especially when working with lambda expressions (anonymous functions). Here are some commonly used functional interfaces and how they can benefit your Java code:

**1. Consumer<T>**

* **What it is:** A Consumer takes a single argument of any type T but doesn't return a value (void).
* **Why we use it:** We use Consumer to perform actions on data or UI elements. It's useful for situations where you want to "consume" or process a value without needing to return anything.

**Example:**

Java

// Printing the length of a String

public class StringConsumer {

public static void main(String[] args) {

String name = "Alice";

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

printStringLength.accept(name); // Output: 5

}

}

Use code [with caution.](https://gemini.google.com/faq#coding)

content\_copy

**2. Predicate<T>**

* **What it is:** A Predicate takes a single argument of any type T and returns a boolean value.
* **Why we use it:** We use Predicate to define conditions for checks and assertions. It allows you to write cleaner and more readable conditional logic.

**Example:**

Java

// Checking if a number is even

public class NumberPredicate {

public static void main(String[] args) {

int number = 10;

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

boolean isEvenNumber = isEven.test(number);

System.out.println(number + " is even: " + isEvenNumber); // Output: 10 is even: true

}

}

Use code [with caution.](https://gemini.google.com/faq#coding)

content\_copy

**3. Function<T, R>**

* **What it is:** A Function takes an argument of type T and returns a value of type R.
* **Why we use it:** We use Function for various tasks like data transformation, performing calculations, or manipulating data based on a specific logic. It provides a flexible way to process and transform data.

**Example:**

Java

// Converting a string to uppercase

public class StringFunction {

public static void main(String[] args) {

String message = "Hello World";

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

String upperCaseMessage = convertToUpperCase.apply(message);

System.out.println(upperCaseMessage); // Output: HELLO WORLD

}

}

Use code [with caution.](https://gemini.google.com/faq#coding)

content\_copy

**4. Supplier<T>**

* **What it is:** A Supplier doesn't take any arguments but returns an object of a specific type T.
* **Why we use it:** We use Supplier to generate data dynamically or provide custom logic for retrieving values. It's helpful when you need to create or access data within your code.

**Example:**

Java

// Generating a random number

public class RandomNumberSupplier {

public static void main(String[] args) {

Supplier<Integer> randomNumberSupplier = () -> (int) (Math.random() \* 100);

int randomNum = randomNumberSupplier.get();

System.out.println("Random number: " + randomNum);

}

}

Use code [with caution.](https://gemini.google.com/faq#coding)

content\_copy

**Benefits of Functional Interfaces:**

* **Improved Readability:** Functional interfaces promote a more concise and declarative style of writing code, making your logic easier to understand.
* **Reusability:** You can create generic functions using these interfaces that can be reused across different parts of your code.
* **Testability:** Functional interfaces often make unit testing of your code easier due to their well-defined behavior.

These are just a few examples of functional interfaces in Java. By understanding and using them effectively, you can write cleaner, more concise, and more maintainable Java code.

import java.util.ArrayList;

import java.util.List;

import java.util.function.Consumer;

import java.util.function.Predicate;

import java.util.function.Function;

import java.util.function.Supplier;

import java.util.stream.Stream;

public class FunctionalStreamExample {

public static void main(String[] args) {

// Supplier to generate a list of random numbers (1-10)

Supplier<List<Integer>> randomNumbersSupplier = () -> {

List<Integer> numbers = new ArrayList<>();

for (int i = 0; i < 10; i++) {

numbers.add((int) (Math.random() \* 10) + 1);

}

return numbers;

};

// Generate the list of random numbers

List<Integer> numbers = randomNumbersSupplier.get();

// Function to double each number

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

// Predicate to check if a number is even

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

// Consumer to print the doubled number

Consumer<Integer> printDouble = num -> System.out.println(num);

// Stream the list of numbers

Stream<Integer> numberStream = numbers.stream();

// Apply transformations and filtering

numberStream

.map(doubler) // Double each number

.filter(isEven) // Filter only even numbers

.forEach(printDouble); // Print the doubled even numbers

}

}

Use code [with caution.](https://gemini.google.com/faq#coding)

content\_copy

**Explanation:**

1. **Supplier:** We define a Supplier called randomNumbersSupplier that generates a list of 10 random numbers between 1 and 10.
2. **Function:** A Function named doubler takes an integer as input and returns its double.
3. **Predicate:** A Predicate named isEven checks if an integer is even by returning true if the modulo operation by 2 is 0.
4. **Consumer:** A Consumer called printDouble simply prints the received integer value.
5. We use stream() on the generated list to create a stream of integers.
6. The map operation applies the doubler function to each element in the stream, doubling all numbers.
7. The filter operation uses the isEven predicate to filter out odd numbers, keeping only even numbers in the stream.
8. Finally, forEach iterates over the filtered stream and applies the printDouble consumer to each element, printing the doubled even numbers.

This example demonstrates how you can chain together different functional interfaces to achieve complex data processing tasks in a concise and readable manner.