Lambda Expression-:

Lambda Expressions in Java are the same as lambda functions which are the short block of code that accepts input as parameters and returns a resultant value. Lambda Expressions are recently included in Java SE 8.

In Java, Lambda expressions basically express instances of functional interfaces (An interface with a single abstract method is called a functional interface).

lambda operator -> body

There are three Lambda Expression Parameters are mentioned below:

Zero Parameter

Single Parameter

Multiple Parameters

1. Lambda Expression with Zero parameter

() -> System.out.println("Zero parameter lambda");

2. Lambda Expression with Single parameter

(p) -> System.out.println("One parameter: " + p);

3. Lambda Expression with Multiple parameters

(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

Functional Interface

A functional interface can have any number of default methods. Runnable, ActionListener, and Comparable are some of the examples of functional interfaces.

Some Built-in Java Functional Interfaces

Since Java SE 1.8 onwards, there are many interfaces that are converted into functional interfaces. All these interfaces are annotated with @FunctionalInterface. These interfaces are as follows –

Runnable –> This interface only contains the run() method.

Comparable –> This interface only contains the compareTo() method.

ActionListener –> This interface only contains the actionPerformed() method.

Callable –> This interface only contains the call() method.

// Java program to demonstrate Implementation of

// functional interface using lambda expressions

**class** Test {

**public** **static** **void** main(String args[])

    {

        // lambda expression to create the object

**new** Thread(() -> {

            System.out.println("New thread created");

        }).start();

    }

}

@FunctionalInterface annotation is used to ensure that the functional interface can’t have more than one abstract method. In case more than one abstract methods are present, the compiler flags an ‘Unexpected @FunctionalInterface annotation’ message. However, it is not mandatory to use this annotation.

// Java program to demonstrate lambda expressions to

// implement a user defined functional interface.

@FunctionalInterface

**interface** Square {

**int** calculate(**int** x);

}

**class** Test {

**public** **static** **void** main(String args[])

    {

**int** a = 5;

        // lambda expression to define the calculate method

        Square s = (**int** x) -> x \* x;

        // parameter passed and return type must be

        // same as defined in the prototype

**int** ans = s.calculate(a);

        System.out.println(ans);

    }

}

Cosnumer Interface-:

It represents a function which takes in one argument and produces a result. However these kind of functions don’t return any value.

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations as mentioned below:

Consumer-:

The consumer interface of the functional interface is the one that accepts only one argument or a gentrified argument. The consumer interface has no return value. It returns nothing.

the Bi-Consumer interface takes two arguments

Consumer<Integer> consumer = (value) -> System.out.println(value);

Predicate-:

a function that accepts an argument and, in return, generates a boolean value as an answer is known as a predicate.

Bi-Predicate is also an extension of the Predicate functional interface, which, instead of one, takes two arguments, does some processing, and returns the boolean value.

Example

public interface Predicate<T> {

boolean test(T t);

}

Function -:

A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing

@FunctionalInterface

public interface BiFunction<T, U, R>

{

R apply(T t, U u);

.......}

Supplier-:

The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output. This type of functional interface is generally used in the lazy generation of values. Supplier functional interfaces are also used for defining the logic for the generation of any sequence. For example – The logic behind the Fibonacci Series can be generated with the help of the Stream. generate method, which is implemented by the Supplier functional Interface.

@FunctionalInterface

public interface Supplier<T>{

// gets a result

………….

// returns the specific result

…………

T.get();

}

Method Reference-:

In Java 8, a method reference is a concise way to refer to a method without invoking it. It provides a shorthand notation for creating functional interfaces and allows you to pass methods as arguments or assign them to variables.

A method reference can be seen as an alternative syntax for lambda expressions when the lambda expression simply calls an existing method. It simplifies the code and improves readability by directly referencing an existing method by its name.

There are four types of method references in Java 8:

Reference to a static method: ClassName::staticMethodName

Example: Math::max

Reference to an instance method of a particular object: objectReference::instanceMethodName

Example: str::length (where str is a String object)

Reference to an instance method of an arbitrary object of a particular type: ClassName::instanceMethodName

Example: String::toLowerCase

Reference to a constructor: ClassName::new

Example: ArrayList::new

Method references can be used wherever lambda expressions are used, such as functional interfaces and streams. They provide a way to make code more concise and expressive, especially when invoking existing methods.

Example-:

// A sample functional interface (An interface with

// single abstract method

interface FuncInterface {

// An abstract function

void abstractFun(int x);

// A non-abstract (or default) function

default void normalFun() {

System.out.println("Hello");

}

}

class Test {

// Implementation of the abstractFun method using a method reference

public static void printDouble(int x) {

System.out.println(2 \* x);

}

public static void main(String args[]) {

// Create a reference to the printDouble method

FuncInterface fobj = Test::printDouble;

// This calls the printDouble method through the method reference and prints 10.

fobj.abstractFun(5);

}

}

Stream-:

the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

along with examples:

Creating Streams:

From a Collection:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

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

From an Array:

String[] names = {"Alice", "Bob", "Charlie"};

Stream<String> stream = Arrays.stream(names);

Using Stream.of:

Stream<String> stream = Stream.of("Apple", "Banana", "Cherry");

Intermediate Operations:

Filter:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Stream<Integer> evenNumbers = numbers.stream().filter(n -> n % 2 == 0);

Map:

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

Stream<Integer> nameLengths = names.stream().map(String::length);

Sorted:

List<Integer> numbers = Arrays.asList(5, 3, 1, 4, 2);

Stream<Integer> sortedNumbers = numbers.stream().sorted();

Terminal Operations:

forEach:

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

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

Collect:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> evenNumbers = numbers.stream().filter(n -> n % 2 == 0).collect(Collectors.toList());

Reduce:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Optional<Integer> sum = numbers.stream().reduce(Integer::sum);

Parallel Streams:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.parallelStream().reduce(0, Integer::sum);

Stream API Methods:

count:

long count = stream.count();

distinct:

Stream<Integer> distinctStream = stream.distinct();

limit:

Stream<Integer> limitedStream = stream.limit(3);

These examples provide a glimpse into the various capabilities of the Stream API in Java 8. Streams allow you to perform complex data processing operations in a concise and declarative manner.

Streams don’t change the original data structure, they only provide the result as per the pipelined methods.

Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.