**Java Lambda Expressions**

* Lambda Expressions were added in Java 8.
* A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

**Syntax**

* The simplest lambda expression contains a single parameter and an expression:

***parameter* -> *expression***

* To use more than one parameter, wrap them in parentheses:

***(parameter1, parameter2)* -> *expression***

* If the lambda expression needs to return a value, then the code block should have a return statement.

***(parameter1, parameter2)* -> { *code block* }**

**Example**

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

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

numbers.add(5);

numbers.add(9);

numbers.add(8);

numbers.add(1);

numbers.forEach( (n) -> { System.out.println(n); } );

}

}

**Functional Interface**

* In Java, a functional interface is an interface that contains only one abstract method.
* Functional interfaces are a key concept introduced in Java 8 as part of the support for lambda expressions and the functional programming paradigm.
* These interfaces are used to enable the use of lambda expressions to provide the implementation for the single abstract method.

Key characteristics of functional interfaces:

1. **Single Abstract Method (SAM):**
   * A functional interface must have exactly one abstract method. This method is often referred to as the single abstract method (SAM).
   * It can have multiple default or static methods, but only one abstract method.
2. **@FunctionalInterface Annotation:**
   * The **@FunctionalInterface** annotation is optional but recommended for a functional interface. It helps ensure that the interface remains a functional interface during the development lifecycle.
3. **Lambda Expressions:**
   * Functional interfaces are designed to be used with lambda expressions, allowing developers to write more concise and expressive code.
4. **Method References:**
   * Method references provide a shorthand notation for using a method as a lambda expression and are often used with functional interfaces.
5. **java. util.function Package:**
   * Java 8 introduced the **java.util.function** package, which includes a variety of predefined functional interfaces, such as **Predicate**, **Consumer**, **Function**, **Supplier**, etc.

**Example**

interface Inter{

public void m1(int n);

}

public class LambdaExpressionpgm1 {

public static void main(String[] args) {

Inter i=(n1)->{

System.*out*.println("Executing myAbstractMethod"+n1);

};

i.m1(10);//call

}

}

**Predefined Functional Interface**

* A predefined functional interface in Java is an interface provided by the Java API that defines a single abstract method (SAM) and is intended to be used with lambda expressions or method references.
* These interfaces are part of the java.util.function package introduced in Java 8 to support functional programming features.

**some commonly used predefined functional interfaces:**

1. **Predicate<T>:**

Represents a predicate (boolean-valued function) of one argument.

**Eg:**

**Predicate<Integer> isPositive = num -> num > 0;**

1. **Function<T, R>:**

Represents a function that takes one argument and produces a result.

**Eg:**

**Function<Integer, String> convertToString = num -> "Number: " + num;**

**3.** **Supplier<T>:**

Represents a supplier of results. It has no input arguments and produces a result.

**Eg:**

**Supplier<Double> getRandomNumber = () -> Math.random();**

**4. Consumer<T>:**

Represents an operation that accepts a single input argument and returns no result.

**Eg:**

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

**5.UnaryOperator<T>:**

Represents an operation on a single operand that produces a result of the same type as its operand.

**Eg:**

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

**6. BinaryOperator<T>:**

Represents an operation upon two operands of the same type, producing a result of the same type as the operands.

**Eg:**

**BinaryOperator<Integer> add = (num1, num2) -> num1 + num2;**

**Method Reference**

* Method references in Java 8 provide a concise way to express lambda expressions when calling an existing method.
* They allow you to refer to methods or constructors without invoking them.
* Method references can often make your code more readable and maintainable.

**There are several types of method references in Java:**

1. **Static Method Reference:**

* **Syntax:**  *ClassName::staticMethodName*
* **Example:**

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

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

**2. Instance Method Reference on a Particular Instance:**

* **Syntax:** instance::instanceMethodName
* **Example:**

String greeting = "Hello";

Function<String, Integer> lengthFunction = greeting::length;

System.out.println(lengthFunction.apply(greeting));

**3. Instance Method Reference on an Arbitrary Instance of a Particular Type:**

* **Syntax:** ClassName::instanceMethodName
* **Example:**

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

names.sort(String::compareToIgnoreCase);

4. **Constructor Reference:**

* **Syntax:** ClassName::new
* **Example:**

Supplier<List<String>> listSupplier = ArrayList::new;

List<String> emptyList = listSupplier.get();

**Streams**

* Streams in Java provide a powerful and expressive way to process sequences of elements.
* Introduced in Java 8, the Stream API allows developers to perform functional-style operations on collections of data, making code more concise and readable.

**key concepts and examples related to Java Streams:**

1. **Creating Streams:**

* Streams can be created from various sources, including collections, arrays, I/O channels, and generator functions.

**// From a collection**

List<String> myList = Arrays.asList("apple", "banana", "orange");

Stream<String> streamFromList = myList.stream();

**// From an array**

String[] myArray = {"apple", "banana", "orange"};

Stream<String> streamFromArray = Arrays.stream(myArray);

**// From a factory method**

Stream<String> streamGenerated = Stream.of("apple", "banana", "orange");

1. **Intermediate Operations:**

* Intermediate operations are operations that transform a stream into another stream. Examples include **filter**, **map**, **distinct**, **sorted**, and **limit**.

**// Filter elements that start with 'b' and convert to uppercase**

List<String> result = myList.stream()

.filter(s -> s.startsWith("b"))

.map(String::toUpperCase)

.collect(Collectors.toList());

1. **Terminal Operations:**

* Terminal operations are operations that produce a result or a side-effect. Examples include forEach, collect, reduce, count, and anyMatch.

**// Print each element**

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

**// Collect to a list**

List<String> collectedList = myList.stream().collect(Collectors.toList());

**// Count the number of elements**

long count = myList.stream().count();

**4. Parallel Streams:**

* Streams can be processed in parallel to leverage multicore architectures.

**// Process in parallel**

List<String> resultParallel = myList.parallelStream()

.filter(s -> s.startsWith("a"))

.map(String::toUpperCase)

.collect(Collectors.toList());

**5. Optional with Streams:**

* Streams can work with **Optional** to handle cases where a value might be absent.

Optional<String> firstElement = myList.stream().findFirst();

firstElement.ifPresent(System.out::println);

1. **Stream API for Collections:**

* The Collection interface in Java includes several methods that return streams, making it easy to work with collections.

List<String> filteredList = myList.stream()

.filter(s -> s.length() > 5)

.collect(Collectors.toList());