The Java Stream API provides a powerful way to process sequences of elements. Here are some of the key methods available in the Stream API:

**Stream Creation**

* **stream()**: Creates a sequential stream from a collection.
* **parallelStream()**: Creates a parallel stream from a collection.

**Intermediate Operations**

These operations transform a stream into another stream and are lazily executed:

* **filter(Predicate<? super T> predicate)**: Filters elements based on a condition.
* **map(Function<? super T, ? extends R> mapper)**: Transforms elements using a function.
* **flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)**: Transforms each element into a stream and flattens the result.
* **distinct()**: Removes duplicate elements.
* **sorted()**: Sorts elements in natural order.
* **sorted(Comparator<? super T> comparator)**: Sorts elements using a comparator.
* **peek(Consumer<? super T> action)**: Performs an action on each element.
* **limit(long maxSize)**: Limits the number of elements.
* **skip(long n)**: Skips the first n elements.

**Terminal Operations**

These operations produce a result or a side-effect and trigger the processing of the stream:

* **forEach(Consumer<? super T> action)**: Performs an action on each element.
* **forEachOrdered(Consumer<? super T> action)**: Performs an action on each element in encounter order.
* **toArray()**: Converts the stream to an array.
* **reduce(BinaryOperator<T> accumulator)**: Reduces the elements using an accumulator.
* **collect(Collector<? super T, A, R> collector)**: Collects elements into a collection.
* **min(Comparator<? super T> comparator)**: Finds the minimum element.
* **max(Comparator<? super T> comparator)**: Finds the maximum element.
* **count()**: Counts the number of elements.
* **anyMatch(Predicate<? super T> predicate)**: Checks if any element matches a condition.
* **allMatch(Predicate<? super T> predicate)**: Checks if all elements match a condition.
* **noneMatch(Predicate<? super T> predicate)**: Checks if no elements match a condition.
* **findFirst()**: Finds the first element.
* **findAny()**: Finds any element.

**Specialized Streams**

* **IntStream**, **LongStream**, **DoubleStream**: Specialized streams for primitive types.

These methods allow you to perform complex data processing tasks in a concise and readable manner [[1]](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) [[2]](https://www.geeksforgeeks.org/java-8-stream-tutorial/) [[3]](https://www.baeldung.com/java-8-streams).

Do you have a specific task in mind that you're trying to accomplish with the Stream API?

**References**

[1] [Stream (Java Platform SE 8 ) - Oracle Help Center](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)

[2] [Java 8 Stream Tutorial - GeeksforGeeks](https://www.geeksforgeeks.org/java-8-stream-tutorial/)

[3] [The Java Stream API Tutorial - Baeldung](https://www.baeldung.com/java-8-streams)

Lambda expressions in Java, introduced in Java 8, provide a concise way to represent instances of single-method interfaces (functional interfaces). Here are some key operations and concepts related to lambda expressions:

**Basic Syntax**

A lambda expression consists of three parts:

1. **Argument List**: Parameters for the lambda expression.
2. **Arrow Token (->)**: Separates the parameter list and the body.
3. **Body**: The code to be executed.

Example:

(parameters) -> expression

**Examples of Lambda Expressions**

1. **No Parameters**:

() -> System.out.println("Hello, World!");

1. **Single Parameter**:

(x) -> x \* x

1. **Multiple Parameters**:

(a, b) -> a + b

1. **Block of Code**:

(int x, int y) -> {

int sum = x + y;

return sum;

}

**Common Uses of Lambda Expressions**

1. **Iterating over a Collection**:

List<String> list = Arrays.asList("a", "b", "c");

list.forEach(item -> System.out.println(item));

1. **Filtering a Collection**:

List<String> list = Arrays.asList("a", "b", "c");

list.stream()

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

.forEach(System.out::println);

1. **Sorting a Collection**:

List<String> list = Arrays.asList("b", "a", "c");

list.sort((s1, s2) -> s1.compareTo(s2));

**Functional Interfaces**

Lambda expressions are used to implement functional interfaces. Some common functional interfaces in Java are:

* **Predicate<T>**: Represents a boolean-valued function of one argument.
* **Function<T, R>**: Represents a function that accepts one argument and produces a result.
* **Consumer<T>**: Represents an operation that accepts a single input argument and returns no result.
* **Supplier<T>**: Represents a supplier of results.

**Example of Using a Functional Interface**

@FunctionalInterface

interface MyFunctionalInterface {

void myMethod();

}

public class LambdaExample {

public static void main(String[] args) {

MyFunctionalInterface myFunc = () -> System.out.println("Lambda Expression Example");

myFunc.myMethod();

}

}

Lambda expressions make your code more readable and concise, especially when working with collections and functional interfaces [[1]](https://www.geeksforgeeks.org/lambda-expressions-java-8/) [[2]](https://www.w3schools.com/java/java_lambda.asp) [[3]](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html).

Is there a specific scenario or example you're interested in using lambda expressions for?

**References**

[1] [Java Lambda Expressions - GeeksforGeeks](https://www.geeksforgeeks.org/lambda-expressions-java-8/)

[2] [Java Lambda Expressions - W3Schools](https://www.w3schools.com/java/java_lambda.asp)

[3] [Lambda Expressions (The Java™ Tutorials > Learning the Java Language ...](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)

Creating a custom functional interface in Java is straightforward. A functional interface is an interface with a single abstract method, which can be implemented using a lambda expression. Here's how you can create and use a custom functional interface:

**Step-by-Step Guide**

1. **Define the Functional Interface**:
   * Use the @FunctionalInterface annotation to indicate that the interface is intended to be a functional interface. This annotation is optional but helps catch errors early by ensuring the interface has only one abstract method.
   * Define a single abstract method in the interface.

@FunctionalInterface

public interface MyFunctionalInterface {

void myMethod();

}

1. **Implement the Functional Interface Using a Lambda Expression**:
   * You can implement the functional interface using a lambda expression wherever an instance of the interface is required.

public class LambdaExample {

public static void main(String[] args) {

MyFunctionalInterface myFunc = () -> System.out.println("Hello from Lambda!");

myFunc.myMethod();

}

}

**Example with Parameters and Return Type**

Here's an example of a functional interface with parameters and a return type:

1. **Define the Functional Interface**:

@FunctionalInterface

public interface Calculator {

int calculate(int a, int b);

}

1. **Implement the Functional Interface Using a Lambda Expression**:

public class CalculatorExample {

public static void main(String[] args) {

Calculator add = (a, b) -> a + b;

Calculator multiply = (a, b) -> a \* b;

System.out.println("Addition: " + add.calculate(5, 3)); // Output: Addition: 8

System.out.println("Multiplication: " + multiply.calculate(5, 3)); // Output: Multiplication: 15

}

}

**Key Points to Remember**

* The @FunctionalInterface annotation is not mandatory but is a good practice.
* A functional interface can have multiple default or static methods, but only one abstract method.
* Lambda expressions provide a clear and concise way to implement functional interfaces.

Creating custom functional interfaces allows you to leverage the power of lambda expressions for more readable and maintainable code [[1]](https://www.tutorialspoint.com/how-to-create-our-own-custom-functional-interface-in-java) [[2]](https://www.geeksforgeeks.org/java-functional-interfaces/) [[3]](https://stackoverflow.com/questions/58713190/how-to-use-a-custom-functional-interface-with-a-method-taking-2-arguments-in-an).

Do you have a specific use case in mind for your custom functional interface?

**References**

[1] [Create Custom Functional Interface in Java - Online Tutorials Library](https://www.tutorialspoint.com/how-to-create-our-own-custom-functional-interface-in-java)

[2] [Java Functional Interfaces - GeeksforGeeks](https://www.geeksforgeeks.org/java-functional-interfaces/)

[3] [java - How to use a custom functional interface with a method taking 2 ...](https://stackoverflow.com/questions/58713190/how-to-use-a-custom-functional-interface-with-a-method-taking-2-arguments-in-an)