# Java 8 Feature[Important Only]

## Lambda Expressions in Java

## 1. Concept

### What is a Lambda Expression?

A **Lambda Expression** is a concise way to represent an **anonymous function** (a function without a name) that can be passed as a parameter or assigned to a variable. It allows writing more concise and functional-style code, introduced in **Java 8**.

### Why Do We Need Lambda Expressions?

1. **Simplifies Code**: Reduces the verbosity of anonymous inner classes.
2. **Functional Programming**: Enables functional-style programming in Java.
3. **Improves Readability**: Code becomes cleaner and easier to understand.
4. **Reusability**: Functions can be reused as behavior without wrapping them into full classes.

## 2. How Does It Work?

### Syntax

The syntax of a lambda expression is:

```
(parameters) -> expression
(parameters) -> { statements; }
```

### Key Components

1. **Parameters**: Represent the input to the lambda expression (can be omitted if there is a single parameter).
2. **Arrow Token (`>`)**: Separates parameters from the body.
3. **Body**: The logic of the lambda, which can be a single expression or a block of statements.

### Example:

```
(int a, int b) -> a + b
```

---

## 3. Detailed Example

### Using Anonymous Inner Class

```
import java.util.ArrayList;
import java.util.List;

public class AnonymousInnerClassExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Using Anonymous Inner Class
        names.forEach(new java.util.function.Consumer<String>
() {
            @Override
            public void accept(String name) {
                System.out.println(name);
            }
        });
    }
}
```

### Using Lambda Expression

```
import java.util.ArrayList;
import java.util.List;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Using Lambda Expression
        names.forEach(name -> System.out.println(name));
    }
}
```

---

## 4. Explanation of Lambda Code

1. **Before (Verbose Code)**:
    - An anonymous inner class is created to implement the `Consumer` functional interface.
    - This is verbose and requires additional boilerplate code.
2. **After (Concise Code)**:
    - The lambda expression directly passes behavior (`name -> System.out.println(name)`) as an argument.

- Eliminates the need for boilerplate code like `new Consumer<String>()`.

## 5. Key Use Cases

## 1. Functional Interfaces

Lambda expressions work only with functional interfaces (interfaces with a single abstract method).

**Example: Functional Interface**

```
@FunctionalInterface
interface Greeting {
    void sayHello(String name);
}

public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        // Using Lambda
        Greeting greeting = name -> System.out.println("Hello, " + name);
        greeting.sayHello("Alice");
    }
}
```

## 2. Collections

Lambdas simplify operations on collections.

**Example: Filtering a List**

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
```

```
public class FilterExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);

        // Using Stream with Lambda
        List<Integer> evenNumbers = numbers.stream()
                                           .filter(num -> num
% 2 == 0)
                                           .collect(Collector
s.toList());

        System.out.println("Even Numbers: " + evenNumbers);
    }
}
```

## 3. Custom Sorting

Lambdas are useful for custom sorting using `Comparator`.

**Example: Sorting Names by Length**

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SortingExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Custom Sorting
        Collections.sort(names, (name1, name2) -> name1.lengt
h() - name2.length());

        System.out.println("Sorted by Length: " + names);
    }
}
```

---

# 6. Advanced Lambda Features

## Method References

A **method reference** is a shorthand for a lambda that calls a specific method. It is represented by `ClassName::methodName`.

**Example:**

```
import java.util.Arrays;

public class MethodReferenceExample {
    public static void main(String[] args) {
        String[] names = {"Alice", "Bob", "Charlie"};

        // Using Method Reference
        Arrays.stream(names).forEach(System.out::println);
    }
}
```

## Capturing Variables

Lambdas can capture local variables (effectively final).

**Example:**

```
public class VariableCaptureExample {
    public static void main(String[] args) {
        String greeting = "Hello";

        Runnable runnable = () -> System.out.println(greetin
g); // Captures 'greeting'
        runnable.run();
    }
}
```

## 7. Advantages of Lambda Expressions

1. **Conciseness**: Eliminates boilerplate code.
2. **Improved Readability**: Code is cleaner and easier to understand.
3. **Functional Programming**: Enables a declarative coding style.
4. **Compatibility**: Works seamlessly with Java's existing APIs like `Streams`.

## 8. Limitations of Lambda Expressions

1. **Single Abstract Method**: Works only with functional interfaces.
2. **Readability with Complex Logic**: Lambdas with complex bodies may reduce readability.
3. **Debugging**: Debugging inside a lambda expression can be challenging.

# Functional Interfaces in Java

## What is a Functional Interface?

A **functional interface** in Java is an interface that contains exactly one abstract method. Functional interfaces are used to represent a single functionality and are primarily intended for lambda expressions and method references.

- **Definition**: An interface with one and only one abstract method is called a functional interface.
- **Purpose**: To enable functional programming in Java by using lambda expressions to represent instances of functional interfaces.
- **Key Points**:
    - Annotated with `@FunctionalInterface` (optional but recommended for clarity).
    - Can have default and static methods, but only one abstract method.

---

## Why Functional Interfaces?

1. **Lambda Support**: Enables lambda expressions, making code concise and readable.
2. **Improved Readability**: Reduces boilerplate code in comparison to anonymous classes.
3. **Standardized Patterns**: Simplifies implementation of common operations like filtering, mapping, consuming, or supplying data.
4. **Built-In Functional Interfaces**: Java 8 introduced many functional interfaces in the `java.util.function` package for common use cases.

---

## Concept and Example of Functional Interfaces

### Example: Custom Functional Interface

```
@FunctionalInterface
interface MyFunctionalInterface {
    void display(String message);
}

public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        // Using a lambda expression to implement the interfa
ce
```

```
        MyFunctionalInterface example = (message) -> System.o
ut.println(message);
        example.display("Hello, Functional Interface!");
    }
}
```

**Explanation**:

- `MyFunctionalInterface` has a single abstract method `display`.
- A lambda expression `(message) -> System.out.println(message)` is used to implement it.

---

## Common Functional Interfaces

Java provides several built-in functional interfaces in the `java.util.function` package. Let's explore the commonly used ones:

---

## 1. Predicate<T>

- **Definition**: Represents a function that takes one argument and returns a boolean value.
- **Purpose**: Used for filtering or testing conditions.
- **Abstract Method**: `boolean test(T t)`

**Example: Filtering Even Numbers**

```
import java.util.function.Predicate;
import java.util.Arrays;
import java.util.List;

public class PredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> isEven = (number) -> number % 2 ==
0;
```

```
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5,
6);
        numbers.stream()
               .filter(isEven) // Apply the Predicate
               .forEach(System.out::println); // Output: 2,
4, 6
    }
}
```

**Explanation**:

- The `isEven` Predicate checks whether a number is even.
- Used with `filter` to process a stream of numbers.

---

## 2. Function<T, R>

- **Definition**: Represents a function that takes one argument of type `T` and returns a result of type `R`.
- **Purpose**: Used for data transformation.
- **Abstract Method**: `R apply(T t)`

**Example: Transforming Strings to Uppercase**

```
import java.util.function.Function;
import java.util.Arrays;
import java.util.List;

public class FunctionExample {
    public static void main(String[] args) {
        Function<String, String> toUpperCase = (input) -> inp
ut.toUpperCase();

        List<String> names = Arrays.asList("alice", "bob", "c
harlie");
        names.stream()
             .map(toUpperCase) // Apply the Function
             .forEach(System.out::println); // Output: ALICE,
BOB, CHARLIE
    }
}
```

**Explanation**:

- The `toUpperCase` Function transforms a string to uppercase.
- Used with `map` to process each element in the list.

---

## 3. Consumer<T>

- **Definition**: Represents a function that takes one argument and performs an operation without returning a result.
- **Purpose**: Used for performing actions like logging or printing.
- **Abstract Method**: `void accept(T t)`

**Example: Printing a List**

```
import java.util.function.Consumer;
import java.util.Arrays;
import java.util.List;

public class ConsumerExample {
    public static void main(String[] args) {
        Consumer<String> printName = (name) -> System.out.pri
ntln("Hello, " + name);

        List<String> names = Arrays.asList("Alice", "Bob", "C
harlie");
        names.forEach(printName); // Output: Hello, Alice; He
llo, Bob; Hello, Charlie
    }
}
```

**Explanation**:

- The `printName` Consumer performs an action (printing) for each input.
- Used with `forEach` to process a list of names.

---

## 4. Supplier<T>

- **Definition**: Represents a function that takes no arguments and supplies a result.
- **Purpose**: Used for providing or generating data.
- **Abstract Method**: `T get()`

### Example: Supplying a Random Number

```
import java.util.function.Supplier;
import java.util.Random;

public class SupplierExample {
    public static void main(String[] args) {
        Supplier<Integer> randomNumberSupplier = () -> new Ra
ndom().nextInt(100);

        System.out.println("Random Number: " + randomNumberSu
pplier.get());
    }
}
```

**Explanation**:

- The `randomNumberSupplier` Supplier generates a random number.
- Used with `get()` to retrieve the result.

## Comparison of Functional Interfaces

| Interface | Arguments | Return Type | Use Case |
|---|---|---|---|
| `Predicate<T>` | 1 | `boolean` | Testing conditions or filtering. |
| `Function<T,R>` | 1 | `R` | Transforming data. |
| `Consumer<T>` | 1 | `void` | Performing operations like printing/logging. |
| `Supplier<T>` | 0 | `T` | Supplying or generating data. |

## Streams API in Java

## What is the Streams API?

The **Streams API** in Java, introduced in Java 8, provides a powerful way to process collections of data in a functional and declarative style. It simplifies operations such as filtering, mapping, and reducing collections of data by chaining operations to form pipelines.

## Why Streams API?

1. **Improved Readability**: Simplifies complex data manipulation tasks with functional-style programming.
2. **Efficiency**: Supports parallel processing for better performance.
3. **Flexibility**: Processes data without modifying the underlying source.
4. **Laziness**: Intermediate operations are lazy, meaning they don’t execute until a terminal operation is invoked.

## Concepts of Streams API

- **Stream**: A sequence of elements supporting sequential and parallel operations.
- **Pipeline**: Consists of:
  - **Source**: Input data (e.g., `List`, `Set`, `Map`, arrays).
  - **Intermediate Operations**: Transform or filter the data (e.g., `map`, `filter`).
  - **Terminal Operations**: Produce a result or a side effect (e.g., `collect`, `forEach`).

---

## Example of Streams API

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamsExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        // Using Streams API to filter and transform data
        List<String> filteredNames = names.stream()
                                          .filter(name -> name.startsWith("C"))
                                          .map(String::toUpperCase)
                                          .collect(Collectors.toList());

        System.out.println(filteredNames); // Output: [CHARLIE]
    }
}
```

**Explanation**:

- **Source**: `names` list.
- **Intermediate Operations**:
  - `filter`: Filters names that start with "C".
  - `map`: Converts filtered names to uppercase.
- **Terminal Operation**:
  - `collect`: Collects the result into a list.

---

## Intermediate Operations

**Definition**: Intermediate operations are used to transform a stream, and they are lazy, meaning they are not executed until a terminal operation is applied.

## Common Intermediate Operations

1. `map`: Transforms each element in the stream.
2. `filter`: Filters elements based on a condition.
3. `sorted`: Sorts elements in natural or custom order.

**Code Examples**:

1. `map`:

```
import java.util.Arrays;
import java.util.List;

public class MapExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        List<Integer> squaredNumbers = numbers.stream()
                                               .map(n -> n *
n)
                                              .collect(Collec
tors.toList());
        System.out.println(squaredNumbers); // Output: [1, 4,
9, 16, 25]
    }
}
```

1. `filter`:

```

import java.util.Arrays;
import java.util.List;

public class FilterExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "C
harlie", "David");
        List<String> shortNames = names.stream()
                                       .filter(name -> name.l
ength() <= 3)
                                       .collect(Collectors.to
List());
        System.out.println(shortNames); // Output: [Bob]
    }
}
```

1. `sorted`:

```

import java.util.Arrays;
import java.util.List;
```

```
public class SortedExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 2, 8, 1, 3);
        List<Integer> sortedNumbers = numbers.stream()
                                             .sorted()
                                             .collect(Collectors.toList());
        System.out.println(sortedNumbers); // Output: [1, 2, 3, 5, 8]
    }
}
```

## Terminal Operations

**Definition**: Terminal operations produce a result or a side-effect and trigger the execution of the entire stream pipeline.

## Common Terminal Operations

1. `collect`: Collects the elements of the stream into a collection.
2. `forEach`: Performs an action for each element in the stream.
3. `reduce`: Reduces the stream to a single value by combining elements.

**Code Examples**:

1. `collect`:

```
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class CollectExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "C
harlie", "Alice");
        Set<String> uniqueNames = names.stream()
                                       .collect(Collectors.to
Set());
        System.out.println(uniqueNames); // Output: [Alice, B
ob, Charlie]
    }
}
```

1. `forEach`:

```
import java.util.Arrays;
import java.util.List;

public class ForEachExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        numbers.stream()
               .forEach(System.out::println); // Output: 1,
2, 3, 4, 5
    }
}
```

1. `reduce`:

```
import java.util.Arrays;
import java.util.List;

public class ReduceExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        int sum = numbers.stream()
                         .reduce(0, Integer::sum);
        System.out.println(sum); // Output: 15
    }
}
```

## Method References

- **Definition**: Simplifies lambda expressions by referring to existing methods.
- **Types**:
    - Reference to a static method: `ClassName::staticMethod`
    - Reference to an instance method: `instance::method`
    - Reference to a constructor: `ClassName::new`

**Example**:

```
import java.util.Arrays;

String[] names = { "Alice", "Bob", "Charlie" };
// Using method reference
Arrays.sort(names, String::compareToIgnoreCase);
System.out.println(Arrays.toString(names));
```