GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK
Approved by AICTE & Affiliated to AKTU

Department of Computer Science and Engineering

**Object Oriented Programming with Java**

**(BCS-402)**

# Presentation Report


**Session – 2023-24**
**(Even Semester)**

| **Submitted To - Mr. Jitendra Singh** | **Submitted By -** | |
|---|---|---|
| | **Roll No.** | **Name** |
| **Date of Presentation: 29-05-2025**<br>**Mode of Presentation: Chalk and Board** | **2201920100186** | **Mradu Bajpai** |
| | **2201920100187** | **Nainsi Verma** |
| | **2201920100188** | **Nandini Gaur** |
| | **2201920100189** | **Naveen Gupta** |
| | **2201920100190** | **Neelesh Kumar** |
| | **Section and Group: D(G1)** | |

# Index

# Functional Interface:-

## Introduction

Functional interfaces are a key feature in modern programming languages, particularly those that support functional programming paradigms. A functional interface is an interface that contains only one abstract method. They are pivotal in enabling lambda expressions, method references, and functional programming constructs in languages like Java.

## Definition:-

A functional interface is an interface with just one abstract method, though it can have multiple default or static methods. This single abstract method is the function that is implemented when a lambda expression or method reference is used.

## Characteristics:-

- Single Abstract Method: The defining characteristic of a functional interface is that it has only one abstract method.
- Optional Annotation: In Java, the @FunctionalInterface annotation can be used to indicate that an interface is intended to be functional. While not mandatory, it helps in identifying the interface and provides compile-time checking.
- Lambda Expressions: Functional interfaces are the target types for lambda expressions and method references

## Importance:-

Functional interfaces are crucial in simplifying and reducing boilerplate code, enabling a more readable and concise way of coding. They bridge the gap between object-oriented and functional programming.

## Uses:-

- Lambda Expressions: Used to implement the single abstract method of functional interfaces concisely.
- Method References: Provide a way to refer to methods directly without executing them.
- Streams and Collections: Functional interfaces are widely used in the Java Streams API, enhancing operations on collections with functional-style methods.

## Common Functional Interfaces in Java:-

Java provides several built-in functional interfaces in the java.util.function package:

- **Predicate**

Represents a boolean-valued function of one argument.
Predicate<String> isEmpty = String::isEmpty;

- **Function**

Represents a function that accepts one argument and produces a result.
Function<String, Integer> length = String::length;

- **Consumer**

Represents an operation that accepts a single input argument and returns no result.
Consumer<String> print = System.out::println;

- **Supplier**

Represents a supplier of results.
Supplier<String> supplier = () -> "Hello, World!";

## SAMPLE CODE ON FUNCTIONAL INTERFACE :-

```
@FunctionalInterface
public interface MyFunctionalInterface {
    void execute();

    default void print() {
        System.out.println("Default Method");
    }

    static void display() {
        System.out.println("Static Method");
    }
}
```

# Lambda Expressions:-

## What?

Lambda expressions are a way to write short, anonymous functions in many programming languages.They are often used to define simple functions that are passed as arguments to other functions.This can make code more concise and readable.

Here are some key points about lambda expressions:

- **Anonymous:** They don't have a name, so you define them inline where you use them.
- **Concise:** They are a short way to write simple functions.
- **Parameters:** They can take zero or more parameters.
- **Return values:** They can return a value or a block of code.

## Why?

There are several reasons why lambda expressions are widely used in programming:

- **Concise Code:** They can dramatically reduce the amount of code needed to define simple functions, especially compared to defining anonymous inner classes. This can make your code easier to read and understand.
- **Improved Readability:** For operations on collections (like filtering or sorting), lambda expressions can often make the code much more readable because the logic is placed directly next to the collection being operated on.
- **Functional Programming:** Lambda expressions are a cornerstone of functional programming, where functions are treated like values and can be passed around and stored in variables.This enables a more declarative style of programming.
- **Reduced Boilerplate:** Lambda expressions eliminate the need to create separate classes just to implement a single method interface. This reduces boilerplate code and improves overall code maintainability.

## How?

### Traditional approach (without lambdas):

Imagine you have a list of numbers and you want to print only the even ones. Here's how you might do it without lambda expressions:

1. **Define a separate method:** You'd create a separate method to check if a number is even. This method would likely take a number as input and return a boolean (true if even, false otherwise).
2. **Use the method in a loop:** Then, you'd iterate through the list of numbers using a loop (like a for loop). Inside the loop, you'd call the method you created earlier to check if the current number is even.
3. **Conditional printing:** If the method returns true (meaning the number is even), you'd print the number.

**Here's some example code (Java) to illustrate this approach**:

```
 public static boolean isEven(int number) {
 return number % 2 == 0;
 }

 public static void main(String[] args) {
 List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
 for (int number : numbers) {
    if (isEven(number)) {
    System.out.println(number);
  }
 }
}
```

**Using lambda expressions:**

With lambda expressions, you can achieve the same functionality in a more concise way:

1. **Inline function:** The lambda expression acts as an anonymous function defined directly within the loop. It takes a number as input and returns a boolean indicating if it's even.
2. **Filtering:** The loop can leverage a built-in method like **forEach** that can iterate through the list and execute the lambda expression for each number.

Here's the code using a lambda expression:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.forEach(number -> {
 if (number % 2 == 0) {
   System.out.println(number);
 }
});
```

In this example, the lambda expression **number -> number % 2 == 0** replaces the separate **isEven** method. It's more concise and keeps the logic focused on the even number check within the loop itself.

This is a simple example, but it demonstrates the benefits of lambda expressions: **reduced code, improved readability, and keeping related logic together.**

# Method References:-

## What?

Java provides a new feature called method reference in Java 8. **Method reference** is used to refer method of functional interface. They allow you to refer to methods or constructors without invoking them. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

- The general syntax of a Method reference
  To refer to a method in an object
  `Object::methodName;`

- Shorthand to print all elements in a list
  To make the code clear and compact, In the above example, one can turn lambda expression into a method reference:
  `list.forEach(System.out::println);`

## Why?

There are several reasons why lambda expressions are widely used in programming:

- Method references improve **code readability and reduce verbosity** by eliminating unnecessary lambda expressions.
- They make the code **more expressive** by directly referencing the method that will be executed.
- They are **compact and easy-to-read** as compared to lambda expressions.
- A method reference is the **shorthand syntax for a lambda expression** that contains just one method call.

## How?

Method references use the `::` operator and come in three main forms:

1. **Reference to a static method:**
   Syntax-
   `ContainingClass::staticMethodName`

Example-

```
import java.util.*;
public class StaticMethodReference{
public static void main(String args[]) {
        List<Integer>list=Arrays.asList(1,2,3,4,5,6,7,8,9,10);
        // Method reference
        list.forEach(StaticMethodReference::print);
    }
       public static void print(final int number){
    System.out.println(number);
  }
}
```

Output-


2. **Reference to an instance method:**
   Syntax-
   `ContainingObject::instanceMethodName`

   Example-

```
import java.util.*;
public class ArbitraryInstanceMethodReference {
    public static void main(String args[]) {
        final List<Person> people = Arrays.asList(new
Person("Anne"), new Person("Lauren"));
        // Method reference
        people.forEach(Person::printName);
    }
    private static class Person {
        private String name;
        public Person(final String name) {
            this.name = name;
        }
        public void printName() {
            System.out.println(name);
        }
    }
}
```

   Output-

```
Anne
Lauren
```

3. **Reference to a constructor.**
   A constructor is referred by using the new keyword.
   Syntax-
   `ClassName::new`

Example-

```
import java.util.function.*;
public class ConstructorReference {

public static void main(String[] args) {

// Constructor Reference
      Function<String, Student> f1 = Student::new;
      System.out.println(f1.apply("Java").getName());
}
}
class Student {
private String name;
         public Student(String name) {
             this.name = name;
         }
      public String getName() {
         return name;
      }
 }
```

Output-

```
Java
```

# Question & Answer:-

### 1.Difference between Interface and Functional Interface.

**Ans.**

A functional interface contains a single abstract method, whereas an ordinary interface can have multiple abstract methods.
A functional interface enable the use of lambda expressions to create more concise and readable code.

### 2.What is the usage of Lambda expression?

**Ans.**

In Java, lambda expressions are a new feature that was introduced in Java 8. They are a concise way to write anonymous functions.
They can also help you to write more functional code.

### 3.Why Method Reference are used if Lambda already exists?

**Ans.**

In Java, method references are a way to reference existing methods and use them instead of lambda expressions.
Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.