# Method References

In Java, both **lambda expressions** and **method references** provide a way to pass functionality as arguments to functions. While lambda expressions are powerful and flexible, **method references** offer a more concise, expressive, and readable alternative when appropriate.

#### 1. ****Conciseness****

* **Method references** allow you to express an operation in a more compact form compared to lambda expressions. Instead of writing out the entire lambda body, you simply reference an existing method.

**Example with Lambda Expression:**

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

**Equivalent with Method Reference:**

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

* Here, the lambda expression s -> System.out.println(s) simply calls the println method, and the method reference System.out::println eliminates the need to explicitly define the lambda. This makes the code shorter and easier to read.

#### 2. ****Improved Readability****

* **Method references** make it clearer that you're simply referring to an existing method rather than defining new behavior inline, as is the case with lambda expressions. This improves **code readability** and **clarity**.

**Example with Lambda Expression:**

list.stream().map(s -> s.length());

**Equivalent with Method Reference:**

list.stream().map(String::length);

* The method reference String::length directly communicates that we are using the length() method from the String class. This makes it easier to understand the intent of the code at a glance, especially for simple operations like calling a method.

#### 3. ****Avoiding Redundancy****

* Method references help you **avoid redundancy** by eliminating the need to write out lambda parameters when they are only used to invoke an existing method. This makes the code less cluttered.

**Example with Lambda Expression:**

list.stream().map(s -> s.toUpperCase());

**Equivalent with Method Reference:**

list.stream().map(String::toUpperCase);

* Here, the lambda expression is simply calling the toUpperCase() method, so the method reference String::toUpperCase is a cleaner and more concise way to express the same operation.

#### 4. ****Aligning with Functional Programming Principles****

* Method references are in line with **functional programming** principles where functions (or methods) are treated as first-class citizens and passed around as arguments. By using method references, you're leveraging the **functional style** of programming in a clear and expressive way.
* When you use a method reference, it signals to the reader that you're invoking a **predefined method** instead of defining a new inline function (as done with lambdas). This can make your code more declarative and easier to maintain.

#### 5. ****When to Use Lambda Expressions Instead of Method References****

* While method references are preferred for simple method invocations, **lambda expressions** are better suited for:
  + Complex operations or custom logic.
  + Inline operations that involve multiple steps, such as conditional statements or calculations.
  + Cases where no existing method is available to use for a method reference.

**Example where Lambda Expression is Better:**

list.forEach(s -> {

if (s.length() > 3) {

System.out.println(s);

}

});

* Here, the lambda expression introduces custom logic (conditional check) that cannot be replaced by a method reference.

### ****Syntax of Method Reference****

The syntax for a method reference is as follows:

ClassName::methodName

This can be broken down as:

* **ClassName**: The name of the class containing the method (it can also be an instance of an object).
* **methodName**: The name of the method being referred to.

### ****Types of Method References****

There are four types of method references in Java:

1. **Reference to a Static Method**
2. **Reference to an Instance Method of a Particular Object**
3. **Reference to an Instance Method of an Arbitrary Object of a Particular Type**
4. **Reference to a Constructor**

## ****Types of Method References with Examples****

### 1. ****Reference to a Static Method****

You can refer to a static method of a class using the class name followed by :: and the method name.

**Example:**

import java.util.Arrays;

import java.util.List;

public class StaticMethodReferenceExample {

public static void main(String[] args) {

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

// Method reference to a static method

names.forEach(System.out::println); // Prints each name using the static println method

}

}

**Explanation:**

* System.out::println is a method reference to the static println() method of the System.out object.

### 2. ****Reference to an Instance Method of a Particular Object****

You can refer to an instance method of a particular object using the syntax object::methodName. This is useful when you have a specific instance of an object and you want to call its method.

**Example:**

import java.util.Arrays;

import java.util.List;

public class InstanceMethodReferenceExample {

public static void main(String[] args) {

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

// Create an instance of a custom class

GreetingService greetingService = new GreetingService();

// Method reference to an instance method

names.forEach(greetingService::greet); // Calls the greet method for each name

}

}

class GreetingService {

public void greet(String name) {

System.out.println("Hello, " + name);

}

}

**Explanation:**

* greetingService::greet is a method reference to the greet() method of the GreetingService class. The method is invoked for each element in the list.

### 3. ****Reference to an Instance Method of an Arbitrary Object of a Particular Type****

This type of method reference allows you to refer to an instance method of any object of a particular type, but you don’t need a specific instance to invoke the method.

**Example:**

import java.util.Arrays;

import java.util.List;

public class ArbitraryObjectMethodReferenceExample {

public static void main(String[] args) {

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

// Method reference to an arbitrary object's instance method

names.forEach(String::toUpperCase); // Calls toUpperCase method on each string in the list

}

}

**Explanation:**

* String::toUpperCase is a method reference to the toUpperCase() method of the String class. This method is invoked for each string element in the list, converting each name to uppercase.

### 4. ****Reference to a Constructor****

You can use method references to refer to a constructor and create new instances of a class. This is useful when you want to create objects in a functional style, especially when working with collections.

**Example:**

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class ConstructorReferenceExample {

public static void main(String[] args) {

// Using constructor reference to create new instances of the Person class

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

List<Person> people = names.stream()

.map(Person::new) // Reference to the constructor of the Person class

.collect(Collectors.toList());

people.forEach(person -> System.out.println(person.getName()));

}

}

class Person {

private String name;

public Person(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**Explanation:**

* Person::new is a constructor reference. It refers to the constructor of the Person class, which takes a String as an argument. The map method applies this constructor to each element in the names list, creating a new Person object for each name.