## 1. Lambda Expressions

### Introduction

Lambda expressions provide a way to express instances of functional interfaces using a concise syntax. They help reduce boilerplate code for anonymous inner classes. They are mainly used to implement simple functionalities without creating new classes or objects.

### Syntax

(parameters) -> expression

(parameters) -> { statements; }

### Example and Explanation

import java.util.\*;

public class LambdaExample {

public static void main(String[] args) {

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

names.forEach(name -> System.out.println(name));

}

}

**Explanation:**

* names is a list containing strings.
* The forEach method takes a lambda expression as an argument.
* The lambda expression name -> System.out.println(name) prints each element.

Lambda expressions are introduced in Java 8 to provide a clear and concise way to express instances of single-method interfaces (functional interfaces). Lambda expressions help reduce boilerplate code and make the code more readable.

**Example: Basic Lambda Expression**

**Step-by-Step Explanation:**

1. Lambda expressions are used to provide the implementation of an abstract method of a functional interface.
2. The syntax of a lambda expression is:

r

Copy code

(parameters) -> expression

java

Copy code

// Functional Interface

interface Greeting {

void sayHello(String message);

}

public class LambdaExample {

public static void main(String[] args) {

// Lambda expression to implement the Greeting interface

Greeting greeting = (message) -> System.out.println("Hello, " + message);

// Using the lambda expression

greeting.sayHello("World");

}

}

**Explanation:**

* The Greeting interface has a single method sayHello(String message).
* The lambda expression (message) -> System.out.println("Hello, " + message) implements the sayHello method.
* When we invoke greeting.sayHello("World"), it prints Hello, World.

**Output:**

Copy code

Hello, World

## 2. Stream API & Operators

### Introduction

Stream API helps process collections of data in a declarative manner using operations such as filtering, mapping, and sorting.

### Example: Filtering and Sorting

import java.util.\*;

import java.util.stream.\*;

public class StreamExample {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);

numbers.stream()

.filter(n -> n > 20)

.map(n -> n \* 2)

.forEach(System.out::println);

}

}

**Explanation:**

* stream() converts the list into a stream.
* filter(n -> n > 20) keeps only numbers greater than 20.
* map(n -> n \* 2) doubles each remaining number.
* forEach(System.out::println) prints the final results.

**Streams API**

The **Streams API** in Java 8 allows us to process collections of objects in a functional style. Streams provide various methods like filter(), map(), forEach(), and others to perform operations on collections in a declarative manner.

**Example: Using Stream API for Filtering and Mapping**

**Step-by-Step Explanation:**

1. Create a stream from a collection (e.g., a List).
2. Use the filter() method to filter elements based on a condition.
3. Use the map() method to transform elements.
4. Use the forEach() method to process the stream.

java

Copy code

import java.util.\*;

import java.util.stream.\*;

public class StreamExample {

public static void main(String[] args) {

// Creating a list of words

List<String> words = Arrays.asList("apple", "banana", "cherry", "date");

// Creating a stream, filtering and mapping

words.stream()

.filter(word -> word.startsWith("b")) // Filter words starting with 'b'

.map(String::toUpperCase) // Convert each word to uppercase

.forEach(System.out::println); // Print each word

}

}

**Explanation:**

* words.stream() creates a stream from the list words.
* .filter(word -> word.startsWith("b")) filters the words that start with the letter "b".
* .map(String::toUpperCase) converts each word to uppercase.
* .forEach(System.out::println) prints each word in uppercase.

**Output:**

Copy code

BANANA

## 3. Functional Interfaces

### Definition

A functional interface is an interface with exactly one abstract method, used mainly for lambda expressions.

### Example and Explanation

@FunctionalInterface

interface MyFunctionalInterface {

void sayHello();

}

public class FunctionalInterfaceExample {

public static void main(String[] args) {

MyFunctionalInterface greet = () -> System.out.println("Hello!");

greet.sayHello();

}

}

**Explanation:**

* MyFunctionalInterface has a single method sayHello().
* A lambda expression implements this interface and prints "Hello!" when invoked.

**Functional Interface**

A **Functional Interface** is an interface that contains only one abstract method, and it may contain multiple default or static methods. Functional interfaces are intended to be implemented by lambda expressions.

**Example: Creating and Using a Functional Interface**

**Step-by-Step Explanation:**

1. Create a functional interface with one abstract method.
2. Implement the interface using a lambda expression.

java

Copy code

@FunctionalInterface

interface Adder {

int add(int a, int b);

}

public class FunctionalInterfaceExample {

public static void main(String[] args) {

// Using a lambda expression to implement the add method

Adder adder = (a, b) -> a + b;

// Using the lambda expression to add two numbers

System.out.println("Sum: " + adder.add(5, 10));

}

}

**Explanation:**

* Adder is a functional interface with the method add(int a, int b).
* The lambda expression (a, b) -> a + b implements the add() method.
* The result of adder.add(5, 10) is printed as the sum of 5 and 10.

**Output:**

makefile

Copy code

Sum: 15

## 4. Default Methods in Interfaces

### Introduction

Default methods in interfaces allow adding new functionality without breaking existing classes implementing the interface.

### Example and Explanation

interface Vehicle {

default void start() {

System.out.println("Vehicle is starting...");

}

}

class Car implements Vehicle {}

public class DefaultMethodExample {

public static void main(String[] args) {

Car myCar = new Car();

myCar.start();

}

}

**Explanation:**

* Vehicle interface has a default method start().
* Car class implements Vehicle but does not override start().
* The default method is invoked when calling start() on Car.

## 5. Method References

### Introduction

Method references provide a way to refer to existing methods without executing them immediately.

### Example and Explanation

import java.util.\*;

public class MethodReferenceExample {

public static void main(String[] args) {

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

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

}

}

**Explanation:**

* Instead of using a lambda name -> System.out.println(name), we use System.out::println to reference the method directly.

**Method References: Complete Explanation with Step-by-Step Example**

**Method References** are a shorthand notation of lambda expressions to call a method directly. They improve code readability by replacing lambda expressions where a method is called on an object or class. They are often used when you want to refer to a method that is already defined, rather than writing out a lambda expression.

In Java 8, you can use method references in the following situations:

* **Static Methods**: Refer to a static method of a class.
* **Instance Methods (on an object)**: Refer to an instance method of a particular object.
* **Instance Methods (on any object of a particular type)**: Refer to an instance method of any object of a given type.
* **Constructor References**: Refer to a constructor of a class.

**1. Static Method Reference**

A static method reference refers to a static method in a class.

**Syntax:**

java

Copy code

ClassName::staticMethodName

**Example: Static Method Reference**

**Step-by-Step Explanation:**

1. Create a class with a static method.
2. Use a method reference to call the static method.

java

Copy code

class Calculator {

// Static method to add two numbers

public static int add(int a, int b) {

return a + b;

}

}

public class MethodReferenceExample {

public static void main(String[] args) {

// Using lambda expression

Operation operation1 = (a, b) -> Calculator.add(a, b);

System.out.println("Sum (using lambda): " + operation1.operation(5, 10));

// Using method reference

Operation operation2 = Calculator::add; // Method reference

System.out.println("Sum (using method reference): " + operation2.operation(5, 10));

}

}

// Functional interface for performing operations

@FunctionalInterface

interface Operation {

int operation(int a, int b);

}

**Explanation:**

* Calculator::add is a **static method reference** that refers to the add method of the Calculator class.
* The method reference simplifies the code by directly pointing to the add method instead of using a lambda expression.

**Output:**

sql

Copy code

Sum (using lambda): 15

Sum (using method reference): 15

**2. Instance Method Reference (on a specific object)**

An instance method reference refers to a method of a particular object.

**Syntax:**

java

Copy code

instanceObject::instanceMethodName

**Example: Instance Method Reference (on a specific object)**

**Step-by-Step Explanation:**

1. Create a class with an instance method.
2. Create an object of the class.
3. Use a method reference to call the instance method.

java

Copy code

class Printer {

// Instance method to print a message

public void printMessage(String message) {

System.out.println(message);

}

}

public class MethodReferenceExample {

public static void main(String[] args) {

Printer printer = new Printer(); // Create an instance of Printer

// Using lambda expression

PrinterAction action1 = (message) -> printer.printMessage(message);

action1.print("Hello, Lambda!");

// Using method reference

PrinterAction action2 = printer::printMessage; // Method reference

action2.print("Hello, Method Reference!");

}

}

// Functional interface to perform an action

@FunctionalInterface

interface PrinterAction {

void print(String message);

}

**Explanation:**

* printer::printMessage is an **instance method reference** that refers to the printMessage method of the printer object.
* The method reference simplifies the code by directly using the method instead of a lambda expression.

**Output:**

mathematica

Copy code

Hello, Lambda!

Hello, Method Reference!

**3. Instance Method Reference (on any object of a particular type)**

An instance method reference can also refer to an instance method of any object of a particular type. This is useful when you're dealing with collections or streams and want to apply the method to each element in the stream.

**Syntax:**

java

Copy code

ClassName::instanceMethodName

**Example: Instance Method Reference (on any object of a particular type)**

**Step-by-Step Explanation:**

1. Create a class with an instance method.
2. Use a method reference in the context of a stream.

java

Copy code

import java.util.Arrays;

import java.util.List;

public class MethodReferenceExample {

public static void main(String[] args) {

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

// Using lambda expression

names.forEach(name -> System.out.println(name.toUpperCase()));

// Using method reference

names.forEach(String::toUpperCase); // Method reference

}

}

**Explanation:**

* String::toUpperCase is an **instance method reference** that refers to the toUpperCase method of the String class. It is called for each element in the names list.

**Output:**

Copy code

ALICE

BOB

CHARLIE

**4. Constructor Reference**

Constructor references refer to the constructor of a class. It is a shorthand for invoking a constructor using a lambda expression.

**Syntax:**

java

Copy code

ClassName::new

**Example: Constructor Reference**

**Step-by-Step Explanation:**

1. Create a class with a constructor.
2. Use a method reference to refer to the constructor.

java

Copy code

class Person {

String name;

// Constructor to initialize name

public Person(String name) {

this.name = name;

}

}

public class MethodReferenceExample {

public static void main(String[] args) {

// Using lambda expression to create a new Person

PersonFactory factory1 = (name) -> new Person(name);

Person person1 = factory1.createPerson("Alice");

System.out.println("Person 1: " + person1.name);

// Using constructor reference

PersonFactory factory2 = Person::new; // Constructor reference

Person person2 = factory2.createPerson("Bob");

System.out.println("Person 2: " + person2.name);

}

}

// Functional interface to create a person

@FunctionalInterface

interface PersonFactory {

Person createPerson(String name);

}

**Explanation:**

* Person::new is a **constructor reference** that refers to the Person class's constructor. This allows us to create a new instance of Person using the createPerson() method of the PersonFactory interface.

**Output:**

yaml

Copy code

Person 1: Alice

Person 2: Bob

**Conclusion**

Method references are a powerful feature introduced in Java 8 that allow developers to write more readable and concise code. They are simply shorthand for lambda expressions where the operation performed is already defined as a method. By using method references, you can refer to static methods, instance methods, and constructors directly.

## 6. Optional Class

### Introduction

Optional is a container object that may or may not contain a non-null value, helping avoid NullPointerException.

### Example and Explanation

import java.util.Optional;

public class OptionalExample {

public static void main(String[] args) {

Optional<String> optional = Optional.ofNullable(null);

System.out.println(optional.orElse("Default Value"));

}

}

**Explanation:**

* Optional.ofNullable(null) creates an empty Optional.
* orElse("Default Value") returns "Default Value" if the Optional is empty.

**What is Optional?**

* **Definition:**  
  Optional is a container object that may or may not contain a value. It helps to deal with null values in a cleaner way by providing methods to handle cases where a value is absent, thus reducing the need for explicit null checks.
* **Why is it useful?**  
  It avoids NullPointerExceptions and helps to make the code more readable and safer. Instead of returning null for missing values, you return an Optional, making it clear that the value might be absent.

**2. Creating Optional Instances**

There are different ways to create Optional instances:

* **Empty Optional:**  
  When there is no value, we use Optional.empty().

java

Copy code

Optional<String> emptyOpt = Optional.empty();

System.out.println(emptyOpt.isPresent()); // Output: false

* **Optional with a non-null value:**  
  Use Optional.of() to wrap a non-null value. If the value is null, it throws a NullPointerException.

java

Copy code

Optional<String> nonNullOpt = Optional.of("Hello");

System.out.println(nonNullOpt.get()); // Output: Hello

* **Optional from a possibly-null value:**  
  Use Optional.ofNullable() when the value might be null. If the value is null, it creates an empty Optional.

java

Copy code

String str = null;

Optional<String> nullableOpt = Optional.ofNullable(str);

System.out.println(nullableOpt.isPresent()); // Output: false

**3. Checking If a Value Is Present**

To check whether the Optional contains a value, we can use these methods:

* **isPresent()**  
  Returns true if the Optional contains a value, and false if it’s empty.

java

Copy code

Optional<String> opt = Optional.of("Java");

System.out.println(opt.isPresent()); // Output: true

* **ifPresent()**  
  Executes a given lambda expression if the value is present, without needing an explicit null check.

java

Copy code

opt.ifPresent(value -> System.out.println("Value is: " + value)); // Output: Value is: Java

**4. Handling Missing Values**

* **orElse()**  
  Returns the value if present; otherwise, returns the default value provided.

java

Copy code

Optional<String> emptyOpt = Optional.empty();

String result = emptyOpt.orElse("Default Value");

System.out.println(result); // Output: Default Value

* **orElseGet()**  
  Similar to orElse(), but the default value is generated lazily by a supplier (i.e., a function).

java

Copy code

String result = emptyOpt.orElseGet(() -> "Generated Default");

System.out.println(result); // Output: Generated Default

* **orElseThrow()**  
  Throws a specified exception if the value is not present.

java

Copy code

Optional<String> emptyOpt = Optional.empty();

try {

emptyOpt.orElseThrow(() -> new IllegalArgumentException("Value is missing"));

} catch (Exception e) {

System.out.println(e.getMessage()); // Output: Value is missing

}

**5. Mapping Values**

You can transform the value inside an Optional using the map() and flatMap() methods.

* **map()**  
  This method applies a function to the value if it’s present. It returns an Optional containing the transformed value.

java

Copy code

Optional<String> nameOpt = Optional.of("Java");

Optional<String> upperNameOpt = nameOpt.map(String::toUpperCase);

System.out.println(upperNameOpt.get()); // Output: JAVA

* **flatMap()**  
  Similar to map(), but the function passed to flatMap() should return an Optional. It’s useful when the transformation itself may return Optional.

java

Copy code

Optional<String> nameOpt = Optional.of("Java");

Optional<String> result = nameOpt.flatMap(name -> Optional.of(name.toUpperCase()));

System.out.println(result.get()); // Output: JAVA

**6. Filtering Values**

The filter() method is used to filter the value inside the Optional.

* **filter()**  
  This method applies a predicate (boolean condition) on the value and returns an Optional:
  + If the condition is met, the Optional remains unchanged.
  + If the condition is not met, an empty Optional is returned.

java

Copy code

Optional<String> nameOpt = Optional.of("Java");

Optional<String> filteredOpt = nameOpt.filter(name -> name.length() > 4);

System.out.println(filteredOpt.get()); // Output: Java

**7. Chaining Optional Methods**

You can chain multiple Optional methods to perform multiple operations.

For example, you can map the value to uppercase and filter it:

java

Copy code

Optional<String> nameOpt = Optional.of("Java");

String result = nameOpt

.map(String::toUpperCase) // Converts the value to uppercase

.filter(s -> s.startsWith("J")) // Filters values starting with "J"

.orElse("Default");

System.out.println(result); // Output: JAVA

**8. Combining Optionals**

You can chain several operations, including flatMap() and filter(), to handle more complex scenarios.

java

Copy code

Optional<String> optionalName = Optional.of("Java");

Optional<String> result = optionalName

.flatMap(name -> Optional.of(name.toUpperCase())) // Converts to uppercase

.flatMap(name -> Optional.of(name + " 8")); // Adds " 8" to the value

System.out.println(result.get()); // Output: JAVA 8

**9. Use Case Example**

Here’s a practical use case where we handle user addresses that may be missing:

java

Copy code

public Optional<String> getUserAddress(String userId) {

// Simulating a user lookup where address may be null

String address = findUserAddress(userId); // May return null

return Optional.ofNullable(address); // Wrap it in an Optional

}

public void displayAddress(String userId) {

Optional<String> addressOpt = getUserAddress(userId);

// Check if value is present and display it

addressOpt.ifPresent(address -> System.out.println("User Address: " + address));

// If value is not present, show a default message

System.out.println(addressOpt.orElse("Address not available"));

}

In this example:

* getUserAddress() wraps the result in an Optional, signaling that the address may be absent.
* displayAddress() either prints the address if present or shows a default message using orElse().

**Summary of Key Methods**

1. **Creation:**  
   Optional.of(), Optional.ofNullable(), Optional.empty()
2. **Presence Check:**  
   isPresent(), ifPresent()
3. **Handling Absence:**  
   orElse(), orElseGet(), orElseThrow()
4. **Transformation:**  
   map(), flatMap()
5. **Filtering:**  
   filter()

## 7. Date and Time API (java.time Package)

### Introduction

Introduced in Java 8, java.time package provides improved handling of dates and times.

### Example and Explanation

import java.time.LocalDate;

import java.time.LocalTime;

import java.time.LocalDateTime;

public class DateTimeExample {

public static void main(String[] args) {

LocalDate date = LocalDate.now();

LocalTime time = LocalTime.now();

LocalDateTime dateTime = LocalDateTime.now();

System.out.println("Date: " + date);

System.out.println("Time: " + time);

System.out.println("Date and Time: " + dateTime);

}

}

**Explanation:**

* LocalDate.now() gets the current date.
* LocalTime.now() gets the current time.
* LocalDateTime.now() gets both date and time.

## 8. Executor Service

### Introduction

ExecutorService provides a way to manage a pool of threads for concurrent execution.

### Example and Explanation

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ExecutorExample {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(2);

executor.execute(() -> System.out.println("Task 1 executed"));

executor.execute(() -> System.out.println("Task 2 executed"));

executor.shutdown();

}

}

**Explanation:**

* Executors.newFixedThreadPool(2) creates a thread pool with two threads.
* execute(() -> System.out.println("Task 1 executed")) runs a task in one of the threads.
* shutdown() stops accepting new tasks and allows existing tasks to complete.

This tutorial provides step-by-step explanations and practical examples for each