Functional Interface (28 Jan)

Functional Interface

A **functional interface** in Java is an interface that has just **one abstract method**, and it may contain multiple default or static methods. These interfaces are primarily used with lambda expressions and method references in Java 8 and later versions.

The main idea behind functional interfaces is to allow instances of interfaces to be created with lambda expressions, providing a more concise and expressive way to write code.

Key characteristics of a functional interface:

- It has exactly one abstract method.
- It can have any number of default and static methods.
- It can be annotated with `@FunctionalInterface` (optional but recommended), which helps the compiler to ensure the interface adheres to the functional interface rules.

> **When you use lambda expressions, you don't have to implement the interface in the `Main` class. The lambda expression itself serves as the implementation of the abstract method, making the code more concise and clean.**

Example:

```
@FunctionalInterface
public interface MyFunctionalInterface {
    void myMethod();  // Single abstract method
```

```
    // Default method (can exist in functional interfaces)
    default void defaultMethod() {
        System.out.println("This is a default method.");
    }

    // Static method (can exist in functional interfaces)
    static void staticMethod() {
        System.out.println("This is a static method.");
    }
}
```

Usage with Lambda Expressions:

```
public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        MyFunctionalInterface myInterface = () -> System.out.println("Lambda expression invoked.");
        myInterface.myMethod();
    }
}
```

In this example, the `MyFunctionalInterface` has one abstract method, `myMethod()`, and can be implemented using a lambda expression. The lambda expression replaces the need to create an anonymous class to implement the interface.

Lambda Expressions

Lambda expressions were introduced in **Java 8** to bring **functional programming** capabilities to the Java language. They allow you to write cleaner, more concise code, especially for functional interfaces.

What is a Lambda Expression?

- A **lambda expression** is a way to define an **anonymous function** (a function without a name).
- It can be used to implement a **functional interface** (an interface with a single abstract method) in a simpler and more concise way.

Syntax of a Lambda Expression

The syntax of a lambda expression is:

```
(parameters) -> expression
```

or

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

Parts of a Lambda Expression:

1. **Parameters**: The input(s) to the function (e.g., `(a, b)` in `(a, b) -> a + b`).
2. **Arrow Token (`->`)**: Separates parameters from the body.
3. **Body**: The logic of the function, which can be a single expression or a block of code.

Examples

1. **Single Parameter Lambda**:

   ```
   x -> x * 2
   ```

   Equivalent to:

   ```
   int multiplyByTwo(int x) {
       return x * 2;
   }
   ```

2. **Multiple Parameters**:

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

   Equivalent to:

   ```
   int add(int a, int b) {
       return a + b;
   }
   ```

3. **No Parameters**:

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

4. **Block Lambda**:

   ```
   (a, b) -> {
       int result = a + b;
       return result * 2;
   }
   ```

Functional Interfaces

A **functional interface** is an interface with a single abstract method. Lambda expressions can be used to provide an implementation for that method.

Examples of Functional Interfaces in Java:

1. **Predefined Functional Interfaces**:
   - `Runnable`: Method `void run()`
   - `Callable`: Method `V call()`
   - `Comparator<T>`: Method `int compare(T o1, T o2)`
   - `Consumer<T>`: Method `void accept(T t)`

- `Supplier<T>`: Method `T get()`
- `Predicate<T>`: Method `boolean test(T t)`
- `Function<T, R>`: Method `R apply(T t)`

2. **Custom Functional Interface**:

```
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}
```

Using Lambda with Functional Interfaces:

```
MathOperation addition = (a, b) → a + b;
System.out.println(addition.operate(5, 3)); // Output: 8
```

Why Use Lambda Expressions?

1. **Conciseness**: Reduce boilerplate code when implementing functional interfaces.
2. **Readability**: Simplify code structure.
3. **Functional Programming**: Enable a functional programming style in Java.
4. **Compatibility**: Use with streams, collections, and functional interfaces in the Java API.

Key Features

1. **Type Inference**:
   - The compiler can infer the types of parameters based on the context.
   - Example:

     ```
     (x, y) → x + y // Compiler infers x and y as int
     ```

2. **Single Statement without `{}`**:
   - For single expressions, you don't need curly braces `{}` or the `return` keyword.
   - Example:

     ```
     (a, b) → a + b
     ```

3. **Block Body for Multiple Statements**:
   - Use `{}` for multiple statements.
   - Example:

     ```
     (a, b) → {
         int sum = a + b;
         return sum * 2;
     }
     ```

4. **Access to Variables**:

- Can access **local variables** (effectively final) and **class fields**.

Where Are Lambda Expressions Used?

1. **Collections API**:
   - Use lambda expressions with methods like `forEach`, `filter`, `map`, and `reduce`.

   Example:

   ```
   List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
   names.forEach(name → System.out.println(name));
   ```

2. **Streams API**:
   - Process collections using streams and lambdas.
     Example:

   ```
   List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
   List<Integer> doubled = numbers.stream()
                               .map(n → n * 2)
                               .collect(Collectors.toList());
   ```

3. **Threading**:
   - Simplify creating threads.
     Example:

   ```
   new Thread(() → System.out.println("Thread running")).start();
   ```

Limitations of Lambda Expressions

1. **Single Abstract Method Only**:
   - Can only be used with functional interfaces.
2. **No Name**:
   - Lambda expressions are anonymous, so debugging might be harder compared to named classes.
3. **Limited Debugging**:
   - You cannot step into a lambda expression in some debuggers.

Behind the Scenes

- Lambda expressions are **syntactic sugar**. At runtime, the compiler converts them into an **anonymous class** or uses the **invokedynamic** bytecode instruction introduced in Java 7.

Advanced Topics

1. **Closures**:
   - Lambda expressions can access **effectively final** local variables:

     ```
     int factor = 2;
     Function<Integer, Integer> multiply = x → x * factor; // `factor` is effectively final
     ```

2. **Method References**:
   - Lambdas can refer to existing methods using method references:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(System.out::println); // Equivalent to name → System.out.println(name)
```

3. **Using Lambdas with Custom Functional Interfaces**:
   - Example:

```
@FunctionalInterface
interface StringOperation {
    String reverse(String input);
}

StringOperation reverse = str → new StringBuilder(str).reverse().toString();
System.out.println(reverse.reverse("Lambda")); // Output: adbmaL
```

Best Practices

1. Use lambdas **only for functional interfaces**.
2. Keep the lambda expression concise and readable.
3. Use **method references** when possible to improve clarity.
4. Avoid complex logic inside lambdas—extract it to a method if needed.
5. Prefer lambdas over anonymous classes for simplicity.

Example: Complete Program

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

public class LambdaExample {
    public static void main(String[] args) {
        // Example: Multiply numbers in a list by 2
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        numbers.stream()
                .map(n → n * 2)
                .forEach(System.out::println);

        // Example: Custom functional interface
        @FunctionalInterface
        interface MathOperation {
            int operate(int a, int b);
        }

        MathOperation addition = (a, b) → a + b;
        System.out.println("Addition: " + addition.operate(5, 3)); // Output: 8
```

```
    }
}
```

Conclusion

Lambda expressions are a powerful feature in Java that enable functional programming, make code concise, and work seamlessly with streams and functional interfaces. Mastering them will significantly improve the quality and readability of your code!

Method Reference Operator

The `::` operator in Java is known as the **method reference operator**. It provides a shorthand way to refer to a method or constructor, allowing you to pass methods as arguments to higher-order functions like those found in the **Stream API** or other functional programming contexts.

Let's break down **method references** from the basics to the example you provided.

1. Basic Concept of Method References:

In Java, a **method reference** is essentially a way to call a method indirectly. It's similar to using a lambda expression, but it's more concise and can make your code easier to read.

2. Lambda Expressions vs Method References:

You might already be familiar with **lambda expressions**. They are a feature in Java that allows you to define functions as arguments for methods or use them inline. Method references are closely related to lambda expressions. Here's a basic comparison:

- **Lambda Expression:**

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

- **Method Reference:**

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

In the example above, both code snippets perform the same action. They print each element in the list. The method reference `System.out::println` is simply a shorthand for the lambda expression `s -> System.out.println(s)`.

3. Types of Method References:

There are four main types of method references in Java:

1. **Reference to a Static Method:**
   - This refers to a **static method** of a class.

```
class MathOperations {
    static int square(int x) {
        return x * x;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Using method reference for a static method
        Function<Integer, Integer> squareFunc = MathOperations::square;
        System.out.println(squareFunc.apply(5)); // Output: 25
    }
}
```

2. **Reference to an Instance Method of a Specific Object:**
   - This refers to an instance method of a specific object.

```
class Printer {
    void print(String message) {
        System.out.println(message);
    }
}

public class Main {
    public static void main(String[] args) {
        Printer printer = new Printer();
        // Using method reference for an instance method
        Consumer<String> printerFunc = printer::print;
        printerFunc.accept("Hello, World!"); // Output: Hello, World!
    }
}
```

3. **Reference to an Instance Method of an Arbitrary Object of a Particular Type:**
   - This refers to a method of **any object** of a specific type.

```
class Person {
    String name;
    Person(String name) {
        this.name = name;
    }
    void greet() {
        System.out.println("Hello, " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        List<Person> people = Arrays.asList(new Person("Alice"), new Person("Bob"));
        // Using method reference for an instance method of any object
        people.forEach(Person::greet); // Output: Hello, Alice  Hello, Bob
    }
}
```

4. **Reference to a Constructor:**
   - This refers to a **constructor** of a class.

```
class Fruit {
    String name;

    Fruit(String name) {
        this.name = name;
        System.out.println("Created fruit: " + this.name);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an ArrayList of fruit names manually using add()
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
        fruits.add("Orange");
        fruits.add("Pineapple");

        // Using method reference for constructor to create Fruit objects
        fruits.forEach(Fruit::new);  // Fruit::new is a reference to the constructor of Fruit class
    }
}
```

4. Detailed Breakdown of Your Example:

Let's look at the example you provided:

```
import java.util.ArrayList;

class Fruit {
    String name;

    // Constructor to initialize the fruit name
    Fruit(String name) {
        this.name = name;
        System.out.println("Created fruit: " + this.name);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an ArrayList of fruit names manually using add()
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
        fruits.add("Orange");
        fruits.add("Pineapple");

        // Using method reference for constructor to create Fruit objects
```

```
        fruits.forEach(Fruit::new);  // Fruit::new is a reference to the constructor of Fruit class
    }
}
```

Here's how it works:

1. **ArrayList Creation:**

   We create an `ArrayList<String>` and add fruit names to it using the `add()` method. This is manually adding elements without using `Arrays.asList()`.

2. **Method Reference to Constructor:**

   `fruits.forEach(Fruit::new);`

   - The `forEach()` method takes a **consumer function** and applies it to each element in the `fruits` list.
   - `Fruit::new` is a constructor reference. It's shorthand for creating new `Fruit` objects using the `Fruit` constructor that accepts a `String` argument.
   - For each fruit name in the list, the constructor of `Fruit` is called with that name, and a new `Fruit` object is created. The constructor prints a message when each fruit is created.

5. When to Use Method References:

- **When you have a function or method** that matches the expected signature of the method being passed to a higher-order function (like `forEach`, `map`, etc.).
- **When the method or constructor is already defined** and you just need to pass it around for execution. It's a more concise way of passing methods as arguments compared to lambdas.

6. Advantages of Method References:

- **More readable:** Using method references is often more concise and readable than using lambdas, especially when the lambda expression just calls an existing method.
- **Less boilerplate:** They allow you to write less code by eliminating the need for writing full lambda expressions.

Summary:

- The `::` operator is a method reference in Java that is used to refer to methods and constructors.
- You can use it with static methods, instance methods, and constructors, often in functional programming contexts like streams or lambdas.
- It's a concise way to pass a method or constructor as an argument to a higher-order function.

Static method in Interface in Java

Static Methods in Java Interfaces (Made Simple)

In Java, **interfaces** can have **static methods**, just like classes. Here's the simplest way to understand them:

What Are Static Methods in Interfaces?

- A **static method** in an interface belongs to the **interface itself**, not the class that implements the interface.

- It **cannot be overridden** by the implementing class because it's tied to the interface.

Why Use Static Methods in Interfaces?

- To provide a **utility method** that is relevant to the interface but does not depend on any specific object.
- For example, a static method can be used to provide **default behaviors** or **helper functions**.

How to Call a Static Method in an Interface?

- You call it using the **interface name**, not through an object or implementing class:

```
InterfaceName.methodName();
```

Key Points to Remember

1. **No Override:** Static methods in interfaces are not inherited, so implementing classes cannot override them.
2. **Belongs to the Interface:** You don't need to create an object of the class to use the static method. It's called directly using the interface name.
3. **Not Mandatory:** Static methods are optional and are just a way to provide extra functionality.

Example

```
interface MyInterface {
    // Static method
    static void display() {
        System.out.println("Hello from static method in interface!");
    }
}

public class MyClass implements MyInterface {
    public static void main(String[] args) {
        // Call the static method using the interface name
        MyInterface.display();
    }
}
```

Output:

```
Hello from static method in interface!
```

Summary

Static methods in interfaces:

- Are called using the **interface name**.
- Cannot be overridden.
- Are great for utility or helper functions.

They're simple, useful, and powerful for certain situations!

Chronological Evolution of Methods in Java Interfaces

Interfaces in Java have evolved significantly over time to add more flexibility and functionality. Here is the **chronological timeline of interface features and methods**:

---

1. Pre-Java 8: Abstract Methods Only

- Before Java 8, interfaces could only have **abstract methods**.
- These methods:
  - Did not have a body (no implementation).
  - Had to be implemented by any class that implemented the interface.

Example (Before Java 8):

```
interface Animal {
    void sound();  // Abstract method
}
class Dog implements Animal {
    public void sound() {
        System.out.println("Bark!");
    }
}
```

---

2. Java 8: Default and Static Methods

Java 8 introduced **default methods** and **static methods** in interfaces.

Default Methods:

- **Purpose:** To add new methods to interfaces without breaking backward compatibility.
- **Key Points:**
  - They have a body (implementation).
  - Can be overridden by implementing classes.

Example:

```
interface Vehicle {
    void start();
    default void stop() {
        System.out.println("Vehicle stopped.");
    }
}
class Car implements Vehicle {
    public void start() {
        System.out.println("Car started.");
    }
}
```

Static Methods:

- **Purpose:** To define utility methods in interfaces.
- **Key Points:**
  - Belong to the interface, not to any implementing class.
  - Cannot be overridden.
  - Called using the interface name.

Example:

```
interface MathUtil {
    static int add(int a, int b) {
        return a + b;
    }
}
public class Main {
    public static void main(String[] args) {
        System.out.println(MathUtil.add(5, 10)); // 15
    }
}
```

---

3. Java 9: Private and Private Static Methods

Java 9 introduced **private methods** and **private static methods** in interfaces.

Private Methods:

- **Purpose:** To allow code reuse within the interface.
- **Key Points:**
  - Can be used by other default or static methods within the same interface.
  - Not accessible by implementing classes or external code.

Example:

```
interface Logger {
    default void logInfo(String message) {
        log(message, "INFO");
    }
    default void logError(String message) {
        log(message, "ERROR");
    }
    private void log(String message, String level) {
        System.out.println("[" + level + "] " + message);
    }
}
```

Private Static Methods:

- **Purpose:** For shared utility logic within the interface.
- **Key Points:**

- Can only be used by other static methods within the same interface.
- Not accessible by implementing classes.

Example:

```
interface StringUtil {
    static void printLowerCase(String input) {
        System.out.println(lowerCase(input));
    }
    private static String lowerCase(String input) {
        return input.toLowerCase();
    }
}
public class Main {
    public static void main(String[] args) {
        StringUtil.printLowerCase("HELLO"); // hello
    }
}
```

---

4. Java 14+: Sealed Interfaces (Introduced in Java 17)

- **Purpose:** To restrict which classes or interfaces can implement/extend an interface.
- **Key Points:**
  - Useful for defining a limited and well-defined hierarchy.
  - Can specify permitted subclasses using the `permits` keyword.
  - Ensures more controlled inheritance.

Example:

```
sealed interface Shape permits Circle, Rectangle {
    void draw();
}
final class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle.");
    }
}
final class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing Rectangle.");
    }
}
```

Summary Table

Version	Feature	Key Points
Pre-Java 8	Abstract Methods	Only abstract methods, no implementation allowed.

Version	Feature	Key Points
Java 8	Default Methods, Static Methods	Default: Backward compatibility; Static: Utility methods tied to the interface.
Java 9	Private, Private Static Methods	Private: Reusable logic within interfaces; Not accessible to implementing classes.
Java 14+	Sealed Interfaces (Java 17)	Restricts which classes/interfaces can implement the interface for controlled hierarchies.

Key Notes:

- Interfaces have transitioned from being purely abstract to supporting a mix of abstract and concrete methods.
- Each addition aimed at improving flexibility, reducing boilerplate, and maintaining compatibility.
- Java interfaces now act as lightweight alternatives to abstract classes while retaining multiple inheritance capabilities.

FINAL VARIABLE

1. Final Instance Variable:

- Declared in a class and must be initialized before use (either in the constructor or at the declaration).

```
class Car {
    final int speedLimit; // Final instance variable

    // Constructor to initialize final variable
    Car(int limit) {
        speedLimit = limit;
    }

    void showSpeedLimit() {
        System.out.println("Speed Limit: " + speedLimit);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(100); // Initializing final variable
        myCar.showSpeedLimit();  // Output: Speed Limit: 100
    }
}
```

2. Final Local Variable:

- A `final` local variable must be initialized before it is used, and its value cannot change afterward.

```
public class Main {
    void showFinalLocalVariable() {
        final int num = 10;  // Final local variable
        // num = 20; // Error: Cannot reassign final variable
        System.out.println("Number: " + num);
    }

    public static void main(String[] args) {
        Main obj = new Main();
        obj.showFinalLocalVariable();  // Output: Number: 10
    }
}
```

3. Final Static Variable:

- A `final` static variable is shared by all instances of a class and can be initialized only once.

```
class MathConstants {
    static final double PI = 3.14159; // Final static variable

    void showPI() {
        System.out.println("PI: " + PI);
    }
}

public class Main {
    public static void main(String[] args) {
        MathConstants obj = new MathConstants();
        obj.showPI();  // Output: PI: 3.14159
    }
}
```

4. Final Parameter:

- A `final` method parameter cannot be reassigned inside the method.

```
public class Main {
    void printNumber(final int num) {  // Final parameter
        // num = 20; // Error: Cannot reassign final parameter
        System.out.println("Number: " + num);
    }

    public static void main(String[] args) {
        Main obj = new Main();
        obj.printNumber(15);  // Output: Number: 15
    }
}
```

5. Final Reference Variable:

- A `final` reference variable cannot refer to a new object once assigned, but the object's properties can be changed.

```
class Car {
    String model;

    Car(String model) {
        this.model = model;
    }
}

public class Main {
    public static void main(String[] args) {
        final Car myCar = new Car("Tesla");  // Final reference variable
        myCar.model = "BMW";  // Allowed: Changing object's properties
        // myCar = new Car("Audi"); // Error: Cannot assign a new object
        System.out.println("Car model: " + myCar.model);  // Output: Car model: BMW
    }
}
```

Summary:

- **Final Instance Variable:** Initialized in the constructor, cannot be changed.
- **Final Local Variable:** Initialized before use, cannot be reassigned.
- **Final Static Variable:** Shared among all objects, initialized once.
- **Final Parameter:** Cannot be reassigned within the method.
- **Final Reference Variable:** Cannot point to another object, but object's fields can change.

These examples should give you a simple overview of how `final` variables work in Java!

FINAL Class

Key Points on `final` Class in Java:

1. **Prevents Inheritance**:
   - A class declared as `final` cannot be subclassed (inherited).
   - This ensures that the behavior of the class remains intact and is not modified by any subclass.

   ```
   public final class FinalClass {
       // Class content
   }
   // Error: Cannot inherit from final 'FinalClass'
   // class ChildClass extends FinalClass {}
   ```

2. **Immutable Behavior**:
   - Marking a class as `final` is often a design decision when you want to ensure that its behavior does not change through inheritance.

- This is useful for creating classes where the logic should remain consistent and predictable (e.g., utility classes or foundational classes).

3. **Security and Integrity**:
   - It helps in ensuring the security and integrity of critical classes, preventing any unintended changes to their methods or behavior.
   - For example, security-sensitive classes like `String` in Java are `final` to prevent subclassing.
4. **Efficient Performance**:
   - JVM optimizations may be more efficient when working with `final` classes, as the method invocations on these classes are determined at compile time, leading to possible inlining and performance gains.
5. **Cannot Extend or Override**:
   - Subclasses cannot override methods of a `final` class, and a subclass cannot inherit or extend its functionality. This helps avoid issues where a method's implementation in a subclass may conflict with the parent class's method.
6. **Example of Usage**:
   - A `final` class can still have constructors, methods, and fields, but none of these can be overridden or inherited.

```
public final class Car {
    public void start() {
        System.out.println("Car is starting");
    }
}

// This will throw an error
// class ElectricCar extends Car {}
```

7. **When to Use**:
   - Use `final` classes when you need to prevent subclassing to maintain the original behavior or ensure immutability.
   - Common examples include `String`, `Integer`, and `Math` classes in Java, where you wouldn't want subclassing to modify how they function.
8. **Cannot be Abstract**:
   - A `final` class cannot be abstract because it cannot have subclasses. Thus, it is fully implemented and cannot serve as a blueprint for other classes.

Summary:

- A `final` class provides strong guarantees about the immutability and behavior of the class, preventing subclassing, method overriding, and offering potential performance optimizations.
- It is used in scenarios where you want to ensure that the class's functionality remains fixed and unchanged.