Java 8 features -→2014

Default Method

Static Method

Functional Interfaces

Lambda Expression

Method References

Functions

Predicate

Consumer

Supplier

Stream API

Optional Class

Default Method

- Introduced in **Java 8** to **add new methods to interfaces** without breaking existing implementations.
- Allows interface methods to have a **default implementation**.
- Classes that implement the interface **can use the default method as-is or override it**.
- Helps with **backward compatibility** — existing classes don't need to implement new methods.

Can we Override default method?

Yes,We can override our default method inside our implementation class

Can I access default method from 2 different interface?

Yes,we can but should have different method signature

Note:

If your are override default method and u want to access it inside the override default method use the super keyword to call it

If the Method Signature are same use the Interface name with super keyword

```java
public interface A {

    public int print(int a);

    public default void m1(String value) {
        System.out.println("A");

    }
}

public interface B {

    public int test(int a);

    public default void m1(String value) {
        System.out.println("B");

    }
}
```

```java
public class C implements A,B{

    public void m1(String value) {

        A.super.m1(value);
    }
    @Override
    public int test(int a) {
        System.out.println("C test");
        return 0;
    }

    @Override
    public int print(int a) {
        System.out.println("C print");
        return 0;
    }
    public static void main(String[] args) {
        C obj=new C();
        obj.m1("A");
        A obj1=new C();
        obj1.m1("A");
        B obj2=new C();
        obj2.m1(null);

    }

}
```

Static Methods

- Also introduced in **Java 8**.
- Static methods in interfaces belong to the interface itself, **not to instances**.
- Can be called using the interface name.

- Useful for **helper or utility methods related to the interface**

```java
public interface A {

    int a=10;
    public int print(int a);

    public default void m1(String value) {
        System.out.println(a);
    }

    public static void m2() {
        System.out.println("A interface");
    }


}
```

```java
public class C implements A{

    @Override
    public int print(int a) {
        System.out.println(a/2);
        return 0;
    }

    public static void main(String[] args) {
        A obj=new C();
        obj.print(30;
        obj.m1("1000");
        A.m2();
    }

}
```

Difference between default and static method inside interface

Functional Programming

- Take inputs and return outputs without changing anything else.
- Avoid changing data (no side effects).
- Combine these functions to build programs.

```
public static int square(int n) {
    return n * n;
}

public static void main(String[] args) {
    int number = 5;
```

```
    int result = square(number);  // Call the function with input 5

    System.out.println("Square of " + number + " is " + result);

}
```

- The function `square` takes a number and returns its square.
- It does **not** change any outside variable or data — just returns a value.
- This is the core idea of functional programming: **small, pure functions that do one thing**.

How to Achieve Functional Programming in Java

- **Understand and Use Functional Interfaces**
  - Use built-in interfaces like `Function`, `Predicate`, `Consumer`, `Supplier`.
  - Create your own with `@FunctionalInterface` annotation when needed.
- **Use Lambda Expressions**
  - Replace anonymous classes with concise lambda syntax.
  - Write functions inline and pass them as arguments.
- **Use Method References**
  - Simplify lambda expressions by referencing existing methods.
  - Example: `System.out::println`
- **Leverage the Stream API**
  - Use streams to process collections declaratively.
  - Apply operations like `map()`, `filter()`, `reduce()`, `forEach()`.
- **Write Pure Functions**
  - Functions should return the same result for the same input.
  - Avoid side effects — don't modify external variables or state.
- **Prefer Immutability**
  - Use `final` keyword for variables.
  - Avoid modifying objects; instead create new objects.
- **Avoid Shared Mutable State**
  - Avoid variables that can be changed by multiple functions or threads.

- **Use Higher-Order Functions**
  - Write functions that take other functions as arguments or return them.
- **Favor Declarative Style Over Imperative**
  - Focus on *what* you want to achieve, not *how* (avoid explicit loops and mutable state).

Functional Interface:

An interface which contains only one abstract method

1 abstract method

N:default method

N: static method

@FuntionalInterface

What is the Purpose of a **Functional Interface** in Java?

A **Functional Interface** in Java is used to **represent a single abstract behavior (function)**, which can be implemented using a **lambda expression**, **method reference**, or **anonymous class**. It enables **functional programming** features in Java.

Before Java 8, Java didn't support passing methods (functions) as arguments. You had to write **anonymous classes**

- **Simplifies Code:**

With one abstract method, you can write inline implementations without needing a full class or anonymous inner class.

- **Clear Intent:**

The interface clearly represents **one specific action or behavior**. It's easier to understand and use.

Use of Functional Interface

Uses of Functional Interface

1. **Enable Lambda Expressions**
   Functional interfaces are the **target types** for lambda expressions. Without them, you can't use lambdas in Java.

2. **Simplify Code**
   They let you write concise, readable, and less boilerplate code instead of using anonymous classes.
3. **Pass Behavior as Parameter**
   You can pass functions (behavior) as method arguments, making your code more flexible and reusable.
4. **Support Functional Programming**
   Functional interfaces allow Java to support functional programming concepts like higher-order functions.
5. **Improve API Design**
   Many Java APIs (like Streams) use functional interfaces for filtering, mapping, and other operations.

```
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Running...");
    }
});
```

With Functional Interface

```
Thread t = new Thread(() -> System.out.println("Running..."));
```

his is possible **because `Runnable` is a functional interface** (only one abstract method: `run()`)

Interface	Method	Usage Example
`Function<T,R>`	`R apply(T t)`	Transform one type to another
`Predicate<T>`	`boolean test(T t)`	True/false check
`Consumer<T>`	`void accept(T t)`	Takes input, no return
`Supplier<T>`	`T get()`	Supplies data, no input

What is a Lambda Expression in Java?

A **lambda expression** is a **short block of code** (anonymous function) that you can pass around — like a function without a name.

It **implements** a functional interface's single abstract method **in a concise way**.

Single Functionality means a function, method, or class should have **only one reason to change**, meaning it should perform **only one specific task or responsibility**.

- A **function** should do **only one job**.
- A **class** should represent **one concept**.
- A **method** should handle **one operation**.

Benefit	Explanation
Readability	Clear and understandable code
Reusability	Small tasks can be reused
Testability	Easy to write unit tests
Debugging	Easy to locate and fix bugs
Maintainability	Easy to update or extend code safely

```
Function<Integer, Integer> square = x -> x * x;
```

Basic syntax

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

Example:

```
@FunctionalInterface
interface Greeting {
   void sayHello(String name);
}
Greeting greet = (name) -> System.out.println("Hello, " + name);
greet.sayHello("Hari");
```

Why only one abstract method?

Lambda expressions implement exactly one method.
If more than one abstract method exists, lambda is ambiguous.
To enable concise lambda usage, only one abstract method is allowed.

What is a Higher-Order Function?

A **higher-order function** is a function that:

1. **Takes another function as an argument**, or
2. **Returns a function as a result**

This is a **core concept in functional programming**.

```
public interface MyFunction {

	public int square(int x);
}

public class FunctionImp {

	public  void hof(int[] a,MyFunction func) {
		for(int i=0;i<a.length;i++) {
			a[i]=func.square(a[i]);
		}
	}
}

public class Driver {
	public static void main(String[] args) {
		int a[]= {10,20,30};
		new FunctionImp().hof(a, x->x*x);//HOF
		System.out.println(Arrays.toString(a));
	}
}
```

- `MyFunction` is a **custom functional interface** with one abstract method.
- `hof()` is a **higher-order function** because it **accepts another function as an argument**.
- We pass a squaring function (`x -> x * x`) to transform the array.

Predefined Functional Interfaces

1)java.util.function

Method References

MethodReferences are a special type of lambdaExpression

We have to use or utilize the existing method implementation which are suitable or compatible for the functional interface implementation of abstract method

Reusing of existence method

Implementation

Direct Method Call: Your lambda *only calls an existing method* (static, instance, or constructor).
Example: `s -> s.toUpperCase()` → `String::toUpperCase`.

Simplify code for readability (e.g., `Math::max` instead of `(a, b) -> Math.max(a, b)`).

- **Prefer Method References** for simplicity and readability in straightforward cases.
- **Use Lambdas** for flexibility, complex logic, or when method references don't align with the target functional interface.

4 types

1.Refernce to static Methods

ClassName::MethodName

The Method which is referred should be matched to method signature of FI abstract method

```
        public class Employee {

           private int id;
           private String name;

           public static boolean validateEmployee(Employee e) {
                      if(e.getId()>=0 && e.getName()!=null) {
                                 return true;
                      }
                      return false;
           }

           public int getId() {
                      return id;
           }

           public void setId(int id) {
                      this.id = id;
           }

           public String getName() {
                      return name;
           }

           public void setName(String name) {
                      this.name = name;
           }

}
@FunctionalInterface
public interface EmployeeValidation {

           public boolean valid(Employee e);

}
public class Test {
           public static void main(String[] args) {

                EmployeeValidation validEmployee =  Employee()::validateEmployee;
                Employee e=new Employee();
                System.out.println(validEmployee.valid(e));
           }
}
The Method which is reffered should be matched to method signature of FI abstract method
```

```
Example
public boolean validateEmployee(Employee e) {
                    if(e.getId()>=0 && e.getName()!=null) {
                                return true;
                    }
                    return false;
          }
public boolean valid(Employee e);
```

2.Reference to instance Methods

*Create the instance of the class where the method is present

The Method which we are reffering from any class Employee , that method should not contains any method argumnets and the same should contains same return value of abstract method which defined inside Functional Interface.

If the method return type is same as functional interface return type and it a no-argument method

We can access instance method with class name also

3.Refernce to instance Method of arbitrary Object of Given Types

4.Reference to Constructor

Make sure abstract method parameter are part of Constructor Definition w.r.to order and type

Use the constructor (i.e., `new` keyword) to create an object, but write it in a short and clean way

ClassName::new

- You want to create objects using a constructor.
- You are using a functional interface (like `Supplier`, `Function`, etc.)

```
class Student {

    Student() {

        System.out.println("Student object created!");
```

```
    }

}
```

Function Interface

The `Function` interface in Java, part of the `java.util.function` package, is a functional interface used to represent a function that accepts one argument and produces a result. It is commonly used for transforming data, method references, or lambda expressions

- **Package**: `java.util.function.Function`
- **Type Parameters**:
  - `T`: The type of the input to the function.
  - `R`: The type of the result of the function.
- **Abstract Method**:
  - `R apply(T t)`: Applies the function to the given argument and returns a result.
- **Default Methods**:

1. `andThen(Function<? super R, ? extends V> after)`
2. `compose(Function<? super V, ? extends T> before)`

Static Method:

- `identity()`: Returns a function that always returns its input argument.

Example:

```
Function<Integer,Double> halfValue=i->i/2.0;
        Double result=halfValue.apply(20);
        System.out.println(result);

        Function<String, String> func1=String::toUpperCase;

        System.out.println(func1.apply("Allen"));
        System.out.println(func1.apply("king"));

        //null check
        Function<String,String> func2= str->str==null?"Enter Valid   Value":str;
        String res = func2.andThen(func1).apply("King");
        System.out.println(res);

//      res = func2.compose(func1).apply(null);
//      System.out.println(res);
```

```
Function<String,Integer> stringToInt = Integer::parseInt;
System.out.println(stringToInt.apply("10"));
```

Predicate Interface

A **`Predicate`** is a **functional interface** introduced in **Java 8** as part of the `java.util.function` package. It represents a **boolean-valued function** of one argument—commonly used for **filtering or testing conditions**.

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

- Takes **one input argument**.
- Returns a **boolean**.

Method	Description
`test(T t)`	Evaluates the predicate on the input argument.
`and(Predicate other)`	Combines two predicates using logical AND.
`or(Predicate other)`	Combines two predicates using logical OR.
`negate()`	Returns a predicate that represents logical negation.
`isEqual(Object target)`	Static method to check object equality.

```
Predicate<Integer> p1=x->x%2==0;
System.out.println(p1.test(250));
Predicate<String> p2=x->x.startsWith("A");
Predicate<String> p3=x->x.endsWith("x")||x.endsWith("X");

System.out.println(p2.test("Apple"));
System.out.println(p3.test("Ajax"));
//And
```

```
            System.out.println(p2.and(p3).test("bjax"));
            //Or
            System.out.println(p2.or(p3).test("bjax"));
            //Negate
            System.out.println(p2.negate().test("Apple"));

            Predicate<String> p4=Predicate.isEqual("Hello");

            System.out.println(p4.test("Hello"));
```

Example 2:

```
public class Employee<T> {


    private int id;
    private String name;
    private double salary;
    private String gender;


    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }



}
```

```java
public class Driver {
    public static void main(String[] args) {
        Employee e1=new Employee();
        e1.setId(1);
        e1.setName("Allen");
        e1.setSalary(30000);
        e1.setGender("Male");

        Employee e2=new Employee();
        e2.setId(2);
        e2.setName("King");
        e2.setSalary(40000);
        e2.setGender("Male");

        Predicate<Employee> pre1=emp->emp.getSalary()>20000;
        System.out.println(pre1.test(e2));

        Predicate<Employee> pre2=emp->emp.getName().equalsIgnoreCase("Allen");
        System.out.println(pre2.test(e1));

        Predicate<Employee> pre3=emp->emp.getGender().equalsIgnoreCase("male");
        System.out.println(pre3.test(e2));

    }
}
```

`Consumer<T>` Interface

A functional interface in `java.util.function` used for operations that accept a single input and **return no result**. Ideal for side-effects (e.g., printing, modifying data).

Supplier<T>

In Java, the `Supplier` interface is a **functional interface** in the `java.util.function` package. It represents a supplier of results — that is, it **does not take any input but returns a result**.

Use `Supplier` when you want to **generate or supply values without taking any input**. For example:

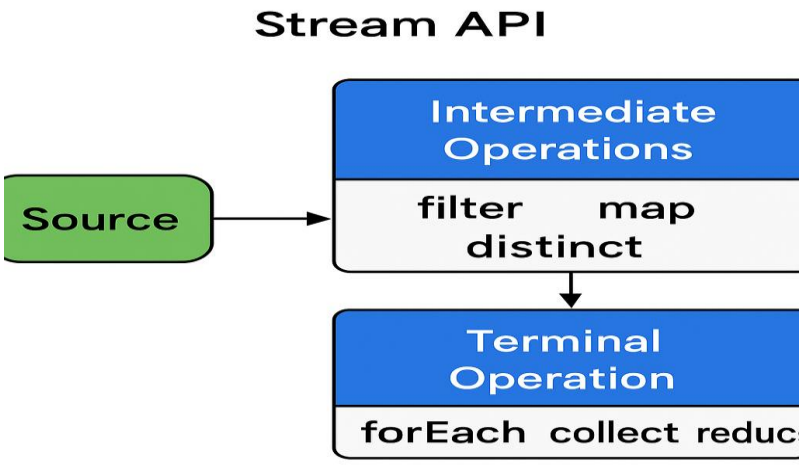
- Generating random numbers
- Supplying default values
- Lazy evaluation (delayed object creation)

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



public class SupplierExample {

    public static void main(String[] args) {

        Supplier<String> supplier = () -> "Hello from Supplier!";

        System.out.println(supplier.get());

    }

}
```

What is a Stream?

A **Stream** is a new abstract layer introduced in Java 8 that allows **processing collections of objects** in a **functional style**.

It simplifies operations such as filtering, mapping, collecting, etc., on data from collections like `List`, `Set`, or arrays.



Features of Stream API

Uses lambda expressions and method references

Intermediate operations are not executed until terminal operation is called

Multiple operations can be chained together

Stream can be consumed only once

Does **not change** the original collection

Stream Pipeline Structure

Source → Intermediate Operation(s) → Terminal Operation

Types of Operations

Intermediate Operations

(These return a new stream)

Method	Purpose
`filter()`	Filter elements based on a condition
`map()`	Transform each element
`sorted()`	Sort the elements
`distinct()`	Remove duplicates
`limit()`	Limit the number of elements
`skip()`	Skip elements

Terminal Operations

(These close the stream)

Method	Purpose
`forEach()`	Perform action for each element
`collect()`	Collect results into a collection
`reduce()`	Reduce the elements to a single value
`count()`	Count elements
`anyMatch()`	Test if any element matches

Program 1: Filter names starting with "A"

```
List<String> names = Arrays.asList("Anu", "Amit", "Ravi", "Asha");

names.stream()
    .filter(name -> name.startsWith("A"))
    .forEach(System.out::println);
```

Program 2: Square even numbers and collect

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
```

```
List<Integer> squares = numbers.stream()
                        .filter(n -> n % 2 == 0)
                        .map(n -> n * n)
                        .collect(Collectors.toList());

System.out.println(squares);
```

Program 3: Count elements greater than 5

```
long count = Arrays.asList(3, 6, 9, 2, 8).stream()
                .filter(n -> n > 5)
                .count();

System.out.println("Count: " + count);
```

Program 4: Convert strings to uppercase

```
List<String> upper = Arrays.asList("java", "stream", "api").stream()
                    .map(String::toUpperCase)
                    .collect(Collectors.toList());

System.out.println(upper);
```

Common Collectors

Collector Method	Purpose
`Collectors.toList()`	Convert stream to `List`
`Collectors.toSet()`	Convert stream to `Set`
`Collectors.joining(", ")`	Join elements into a `String`

Collector Method	Purpose
`Collectors.groupingBy()`	Group elements based on a field

Example: Grouping

```
Map<Integer, List<String>> grouped =
    Arrays.asList("one", "two", "three", "four")
        .stream()
        .collect(Collectors.groupingBy(String::length));


System.out.println(grouped);
```