Lec 47 JAVA 8 P3:

This lesson explains Java 8's functional interfaces and lambda expressions, showing how they simplify code and make programs more efficient. It covers the four main built-in functional interfaces—Predicate, Function, Supplier, and Consumer—explaining their purposes and how to use them with practical examples.

What is a Functional Interface?

A functional interface in Java is an interface with only one abstract method. It is essential for enabling lambda expressions, which allow for concise, readable code by removing the need to write boilerplate method details.

Example: Checking Even Numbers with Functional Interface

To filter even numbers from an array, a functional interface is created with a method to apply custom logic. This method can be implemented using an anonymous inner class, allowing for flexible data processing.

Drawbacks of Custom Functional Interfaces

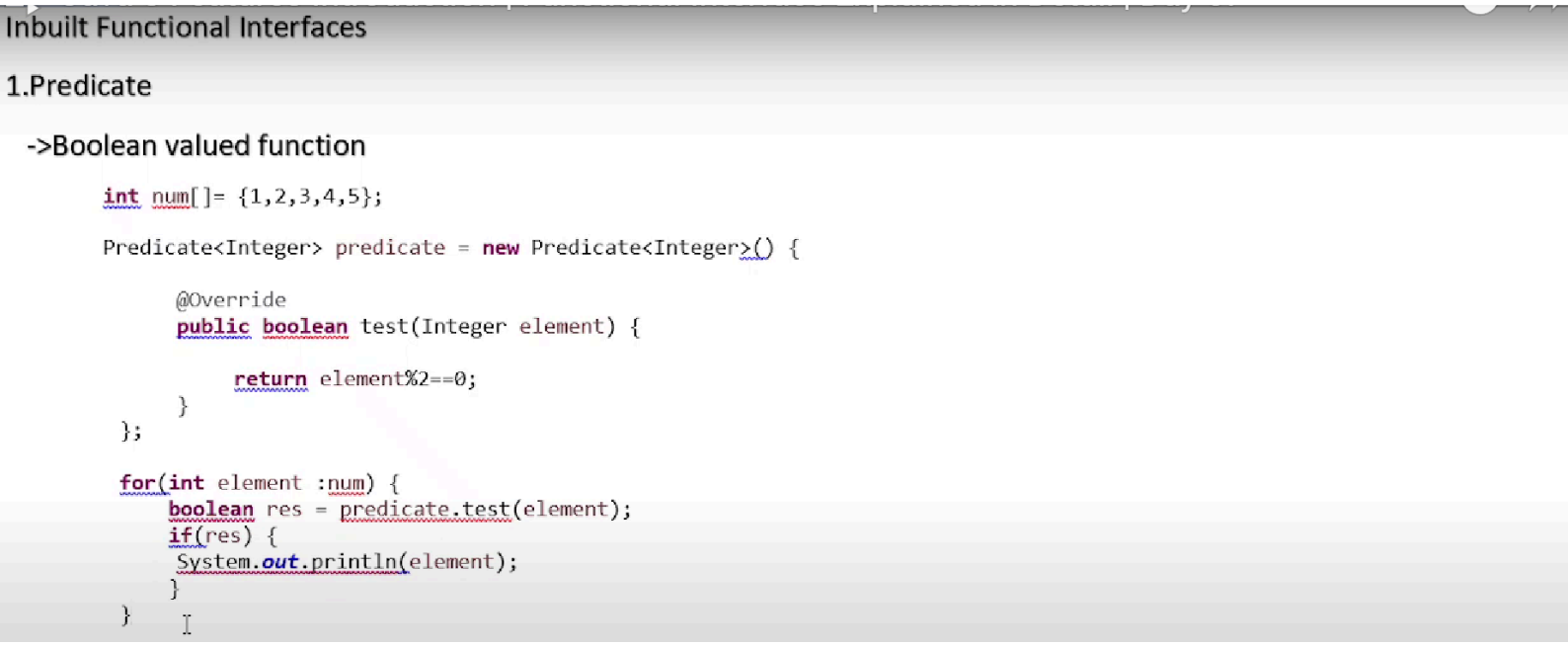
Creating custom functional interfaces repeatedly can be tedious and unnecessary. Java 8 provides built-in functional interfaces to avoid this redundancy and save developer effort.

Four Built-in Functional Interfaces Overview

Java 8 introduces four main functional interfaces: Predicate (for boolean checks), Function (for data transformation), Supplier (for producing values), and Consumer (for processing values without returning anything). These cover most use cases.

Predicate Interface Explained

Predicate is used when a boolean result is needed after applying logic to data. For example, checking if a number is even returns true or false. Predicate can take any data type as input.



Using Built-in Predicate vs Custom Interface

The built-in Predicate interface removes the need to create a new interface for each boolean check, streamlining code and reducing errors. It works similarly to custom interfaces but is standardized and reusable.

Lambda Expressions for Predicate

Lambda expressions allow for the shortest and most readable way to implement Predicate logic. Syntax is simplified by removing class and method declarations, focusing only on the logic itself.

LAMBDA EXPRESSION:

```
int num[]= {1,2,3,4,5};

Predicate<Integer> predicate =(element) ->element%2==0;

for(int element :num) {
    boolean res = predicate.test(element);
    if(res) {
     System.out.println(element);
    }
}
```

Function Interface Explained

The Function interface transforms data from one type to another (or the same type). It is used when input and output types may differ, such as converting a string to its length or to uppercase.

Function Interface Practical Example

By using Function, you can pass a string and get another string or an integer in return, depending on your logic. This makes data transformation flexible and reusable.

2.Function

```
String words[]= {"raju","kaju","aman","arun"};

Function<String, String> fun = new Function<String, String>() {

    @Override
    public String apply(String t) {

        return t.toUpperCase();
    }
};

for(String word  :words) {
    System.out.println(fun.apply(word));
}
```

LAMBDA EXPRESSION:

```
String words[]= {"raju","kaju","aman","arun"};

Function<String, String> fun = (t)-> t.toUpperCase();

for(String word  :words) {
     System.out.println(fun.apply(word));
}
```

Supplier Interface Explained

Supplier is used when you only need to produce or supply a value without any input, like generating a random number or fetching a string. The method has no parameters and only returns a result.

3.Supplier

```
Supplier<String> supplier = new Supplier<String>() {

    @Override
    public String get() {
        //CUSTOM LOGIC
        //Database connection
        //Data retrieve
        //Collection
          return "HELLO CUSTOM";
    }
};

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

LAMBDA EXPRESION

```
Supplier<String> supplier = ()->"HELLO CUSTOM";
 System.out.println(supplier.get());
```

Consumer Interface Explained

Consumer is the opposite of Supplier—it takes input but does not return anything. It's useful for operations like

printing or saving data, where you act on the input but don't need a result.

4.Consumer

```
String words[]= {"rajux","kajuxx","amanxxx","arunxxxx"};

Consumer<String> consumer = new Consumer<String>() {

    @Override
    public void accept(String t) {

        System.out.println(t.length());

    }
};

for(String word  :words) {
    consumer.accept(word);
}
```

LAMBDA EXPRESION

```
String words[]= {"rajux","kajuxx","amanxxx","arunxxxx"};

Consumer<String> consumer =(t)->System.out.println(t.length());

for(String word  :words) {
    consumer.accept(word);
}
```

Efficiency: Anonymous Inner Class vs Lambda Expression

Using anonymous inner classes creates multiple .class files, increasing code size and JVM workload. Lambda expressions, however, reduce code length and store all logic in a single .class file, making programs more efficient and easier for the JVM to handle.

Benefits Beyond Shorter Code

Lambda expressions not only reduce lines of code but also make programs more efficient and enable functional programming styles in Java, promoting clearer and more maintainable code.

Key Takeaways on Lambda Expressions

Lambda expressions do not create anonymous inner classes internally; instead, they provide a unique, efficient implementation. They make code shorter, more efficient, and support modern programming practices in Java.