Lec 46 JAVA 8 P2:

This lesson explains the concepts of anonymous inner classes, functional interfaces, and lambda expressions in Java. It shows how these features help make Java code shorter, easier to manage, and more focused, especially when you only need a specific implementation in one place.

Introduction and Java 8 Feature Overview

The importance of understanding functional interfaces, anonymous inner classes, and lambda expressions is highlighted as key features introduced in Java 8. These concepts are foundational for modern Java programming.

Understanding Anonymous Inner Classes

An anonymous inner class is a class without a name, defined and instantiated in a single statement, often used to provide a specific implementation of an interface or class for one-time use. This avoids creating extra files and keeps code focused when the implementation is needed only once.

When to Use Anonymous Inner Classes

Anonymous inner classes are best used when an interface implementation is needed in only one place, and not reused elsewhere. This reduces unnecessary code and files, making the codebase simpler.

Practical Example: Creating Anonymous Inner Classes

A step-by-step example shows how to define an interface, implement it with an anonymous inner class inside a method, and use it for a specific task (like a bank transaction). This keeps the logic grouped and easy to manage.

Launch.java

```java
package com.mainapp;
public class Launch {

    public static void main(String[] args) {

        //Anonymous inner class
        Bank bank = new Bank() {

            @Override
            public void transaction() {
                System.out.println("Trs Completed");
            }
        };

        bank.transaction();
        //new Bank(); created object of child
        //class(An class) of Bank Interface
    }
}
```

Bank.java

```java
package com.mainapp;

public interface Bank {
    public void transaction();
}
```

Console

```
<terminated> Launch (26) [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (Nov 7, 2024, 9:37:32 PM
Trs Completed
```

Scope and Limitations of Anonymous Inner Classes

Anonymous inner classes are specific to the context where they are created, such as inside a method. They

cannot be reused elsewhere unless explicitly passed around, making them suitable for tightly scoped tasks.

Internal Behavior: Class Files for Anonymous Inner Classes

Each anonymous inner class generates its own .class file, so while code is logically grouped, the Java Virtual Machine (JVM) still manages multiple files internally (if a Launch named java file contains multiple AIC, then it will be named as Launch&1, Launch&2,..... and so on). This does not necessarily make the program more efficient.

Introduction to Functional Interfaces

A functional interface in Java is an interface with only one abstract method. It can have multiple default or static methods, but only one method that must be implemented. This is essential for lambda expressions.

Functional Interface Rules and Annotations

The @FunctionalInterface annotation is optional but helps catch errors at compile time by ensuring only one

abstract method exists. Methods from Object (like toString()) do not count towards this limit.

Built-in Functional Interfaces in Java

Java has several built-in functional interfaces like Runnable and Comparator, which have only one abstract method (run() and compareTo()) and are widely used in multi-threading and collections.

Example: Using Functional Interfaces and Anonymous Inner Classes

Demonstrates how to use a functional interface and an anonymous inner class to check if numbers in an array are divisible by two. The logic is encapsulated and reusable within the method.

```java
package com.mainapp2;
public class Launch {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

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

        Solution solution = new Solution() {

            @Override
            public boolean logic(int element) {

                return element%2==0;
            }
        };

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

```java
package com.mainapp2;

public interface Solution {

    public boolean logic(int element);

}
```

```
<terminated> Launch (27) [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe
2
4
```

- Before this we used to make dedicated class for interface which reused everywhere. But this AIC will provide reusability only in main method (above example).
- We could have simply wrote the logic for even element check for nums array. But by using this Anonymous Inner class, we achieve modularity.
- It is not necessary for the interface of AIC to be functional interface.

Introduction to Lambda Expressions

Lambda expressions are a concise way to write implementations for functional interfaces. They remove unnecessary code by inferring method names, parameter types, and return types from the interface, leading to much shorter and cleaner code.

```java
public static void main(String[] args) {
    // TODO Auto-generated method stub

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

//      Solution solution = new Solution() {
//
//          @Override
//          public boolean logic(int element) {
//
//              return element%2==0;
//          }
//      };


    //LAMBDA EXPRESSION
    Solution solution =(element)->element%2==0;

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

```java
package com.mainapp2;
public interface Solution {

    public boolean logic(int element);

}
```

```
<terminated> Launch (27) [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.e
2
4
```

- In above example, we can see how the code shrinked to single line (lambda expression).

```java
public static void main(String[] args) {
    // TODO Auto-generated method stub

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

//      Solution solution = new Solution() {
//
//          @Override
//          public boolean logic(int element) {
//
//              return element%2==0;
//          }
//      };


    //LAMBDA EXPRESSION
    Solution solution =(element)->element%2==0;

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

```java
package com.mainapp2;
public interface Solution {

    public boolean logic(int element);
    public String logic(String element);

}
```

```
<terminated> Launch (27) [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.ex
2
4
```

- Here, it is necessary for a interface to be a functional interface as lambda expression gets no ambiguity in selecting method from interface.
- But in above code, lambda expression is getting confused whether to logic() with boolean return type

or logic() with String return type.

Lambda Expression Syntax Simplification

If a lambda expression contains only one statement, even curly braces and the return keyword can be omitted, making the code even more compact. Lambda expressions are based on functional interfaces.

Why Lambda Expressions Require Functional Interfaces

Lambda expressions work only with functional interfaces because Java needs to know exactly which method to implement. If an interface has more than one abstract method, the lambda expression cannot be mapped unambiguously.

Java 8 Inbuilt Functional Interfaces

Java 8 provides several inbuilt functional interfaces, like Predicate, so you don't always need to define your own. These can be used directly with lambda expressions for common tasks.

Recap and Importance of Understanding Anonymous Inner Classes

Mastering anonymous inner classes is crucial for understanding lambda expressions and modern Java features. Once comfortable, you can write cleaner and more efficient code using Java 8's capabilities.