JAVA PROGRAMMING

# Week 4:
# Advanced Topics in Classes

Lecturer:

- HO Tuan Thanh, M.Sc.

# Plan



1. Nested classes
2. Local classes
3. Anonymous classes
4. Lambda expressions

# Plan



1. **Nested classes**
2. Local classes
3. Anonymous classes
4. Lambda expressions

# Nested classes



- A nested class is a class defined within another class.
- Nested classes are divided into two categories: non-static and static.
  - Non-static nested classes are called inner classes.
  - Nested classes that are declared static are called static nested classes.

# Nested classes



- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- Static nested classes do not have access to other members of the enclosing class.

# Why use nested classes?



- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- It can lead to more readable and maintainable code.

# Inner classes (Non-static nested classes)



- As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.

- Also, because an inner class is associated with an instance, **it cannot define any static members itself**.

# Inner classes (Non-static nested classes)



```
package package1;

public class OuterClass {
    public class InnerClass {
    }
}
```

```
import package1.*;

public class Main {
    public static void main(String []args){
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new
InnerClass();
    }
}
```

# Static-nested classes



```
package package2;

public class OuterClass {
    public static class InnerClass {
    }
}
```

```
import package2.*;

public class Main {
    public static void main(String []args){
        OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
    }
}
```

# Example



```
package package3;

public class OuterClass {

    private String outerField = "Outer field";
    private static String staticOuterField = "Static outer field";

    class InnerClass {
        void accessMembers() {
            System.out.println(outerField);
            System.out.println(staticOuterField);
        }
    }

    static class StaticNestedClass {
        void accessMembers(OuterClass outer) {
            System.out.println(outer.outerField);
            System.out.println(staticOuterField);
        }
    }
}
```

# Example



```
package package3;

public class OuterClass {
    public static void main(String[] args) {
        System.out.println("Inner class:");
        System.out.println("------------");
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new InnerClass();
        innerObject.accessMembers();

        System.out.println("\nStatic nested class:");
        System.out.println("--------------------");
        StaticNestedClass staticNestedObject = new StaticNestedClass();
        staticNestedObject.accessMembers(outerObject);

        System.out.println("\nTop-level class:");
        System.out.println("--------------------");
        TopLevelClass topLevelObject = new TopLevelClass();
        topLevelObject.accessMembers(outerObject);
    }
}
```

# Example



```
package package3;

public class TopLevelClass {

    void accessMembers(OuterClass outer) {
        // Error!
        System.out.println(outer.outerField);

        // Error!
        System.out.println(OuterClass.staticOuterField);
    }
}
```

# Shadowing



```
package package4;

public class ShadowTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

# Practical example (1/3)



```
package package5;

public class DataStructure {

    // Create an array
    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];

    public DataStructure() {
        // fill the array with ascending integer values
        for (int i = 0; i < SIZE; i++) {
            arrayOfInts[i] = i;
        }
    }

    public void printEven() {

        // Print out values of even indices of the array
        DataStructureIterator iterator = this.new EvenIterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
        System.out.println();
    }
}
```

# Practical example (2/3)



```
package package5;

public class DataStructure {
    interface DataStructureIterator extends java.util.Iterator<Integer> { }

    // Inner class implements the DataStructureIterator interface,
    // which extends the Iterator<Integer> interface

    private class EvenIterator implements DataStructureIterator {

        // Start stepping through the array from the beginning
        private int nextIndex = 0;

        public boolean hasNext() {

            // Check if the current element is the last in the array
            return (nextIndex <= SIZE - 1);
        }

        public Integer next() {

            // Record a value of an even index of the array
            Integer retValue = Integer.valueOf(arrayOfInts[nextIndex]);

            // Get the next even element
            nextIndex += 2;
            return retValue;
        }
    }
}
```

# Practical example (3/3)



```
package package5;

public class DataStructure {
     public static void main(String s[]) {
         // Fill the array with integer values and print out only
         // values of even indices
         DataStructure ds = new DataStructure();
         ds.printEven();
     }
}
```

# Plan



1. Nested classes
2. **Local classes**
3. Anonymous classes
4. Lambda expressions

# Local classes



- Local classes are classes that are defined in a block, which is a group of zero or more statements between balanced braces.
- You typically find local classes defined in the body of a method.

# Example (1/2)



```
package package6;

public class LocalClassExample {
    static String regularExpression = "[^0-9]";
    public static void validatePhoneNumber(String phoneNumber1, String
phoneNumber2) {
        final int numberLength = 10;

        class PhoneNumber {
            String formattedPhoneNumber = null;
            PhoneNumber(String phoneNumber){
                // numberLength = 7;
                String currentNumber = phoneNumber.replaceAll(
                        regularExpression, "");
                if (currentNumber.length() == numberLength)
                    formattedPhoneNumber = currentNumber;
                else
                    formattedPhoneNumber = null;
            }
            public String getNumber() {
                return formattedPhoneNumber;
            }
        }
}
```

# Example (2/2)



```
package package6;

public class LocalClassExample {
    public static void validatePhoneNumber(String phoneNumber1, String
phoneNumber2) {
        final int numberLength = 10;

        PhoneNumber myNumber1 = new PhoneNumber(phoneNumber1);
        PhoneNumber myNumber2 = new PhoneNumber(phoneNumber2);
        if (myNumber1.getNumber() == null)
            System.out.println("First number is invalid");
        else
            System.out.println("First number is " + myNumber1.getNumber());
        if (myNumber2.getNumber() == null)
            System.out.println("Second number is invalid");
        else
            System.out.println("Second number is " + myNumber2.getNumber());
    }

    public static void main(String... args) {
        validatePhoneNumber("123-456-7890", "456-7890");
    }
}
```

# Local classes vs enclosing class' variables



- A local class can only access local variables that are declared final.
- Starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or effectively final.
  - Variable capture.
  - A variable or parameter whose value is never changed after it is initialized is effectively final.
- Starting in Java SE 8, if you declare the local class in a method, it can access the method's parameters.

# Plan



1. Nested classes
2. Local classes
3. **Anonymous classes**
4. Lambda expressions

# Anonymous Classes



- Anonymous classes enable you to make your code more concise.
- They enable you to declare and instantiate a class at the same time.
- They are like local classes except that they do not have a name.
- Use them if you need to use a local class only once.

# Example (1/5)



```
package package10;

public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }
}
```

# Example (2/5)



```
package package10;

public class HelloWorldAnonymousClasses {
    public void sayHello() {

        // local class
        class EnglishGreeting implements HelloWorld {
            String name = "world";
            public void greet() {
                greetSomeone("world");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hello " + name);
            }
        }
        HelloWorld englishGreeting = new EnglishGreeting();

        englishGreeting.greet();
        frenchGreeting.greetSomeone("Fred");
        spanishGreeting.greet();
    }

}
```

# Example (3/5)



```
package package10;

public class HelloWorldAnonymousClasses {
    public void sayHello() {
        // anonymous class
        HelloWorld frenchGreeting = new HelloWorld() {
            String name = "tout le monde";
            public void greet() {
                greetSomeone("tout le monde");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Salut " + name);
            }
        };

        englishGreeting.greet();
        frenchGreeting.greetSomeone("Fred");
        spanishGreeting.greet();
    }
}
```

# Example (4/5)



```
package package10;

public class HelloWorldAnonymousClasses {
    public void sayHello() {
        // anonymous class
        HelloWorld spanishGreeting = new HelloWorld() {
            String name = "mundo";
            public void greet() {
                greetSomeone("mundo");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hola, " + name);
            }
        };
        englishGreeting.greet();
        frenchGreeting.greetSomeone("Fred");
        spanishGreeting.greet();
    }
}
```

# Example (5/5)



```
package package10;

public class HelloWorldAnonymousClasses {
    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp =
                new HelloWorldAnonymousClasses();
        myApp.sayHello();
    }
}
```

# Declaring Anonymous Classes



- Note that you can declare the following in anonymous classes:
  - Fields
  - Extra methods (even if they do not implement any methods of the supertype)
  - Instance initializers
  - Local classes

- However, you cannot declare constructors in an anonymous class.

# Practical example



```
this.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        counter --;
        setText(counter + "");
    }
});
```

# Plan



1. Nested classes
2. Local classes
3. Anonymous classes
4. **Lambda expressions**

# Lamda expressions



- In the final analysis, in much the same way that generics reshaped Java several years ago, lambda expressions continue to reshape Java today.
  - Herbert Schildt, Java: The Complete Reference

# Lamda expressions



- Key to understanding Java's implementation of lambda expressions are two constructs.
  - The first is the lambda expression, itself.
  - The second is the functional interface.

# Lamda expressions



- A lambda expression is, essentially, an anonymous (that is, unnamed) method.
- However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface.
- Thus, a lambda expression results in a form of anonymous class.
- Lambda expressions are also commonly referred to as closures.
- A functional interface is an interface that contains one and only one abstract method.

# Lamda expressions



- The arrow operator, is −>. It divides a lambda expression into two parts.
- The left side specifies any parameters required by the lambda expression.
- On the right side is the lambda body, which specifies the actions of the lambda expression.

# Lamda expressions



- Java defines two types of lambda bodies.
- One consists of a single expression, and the other type consists of a block of code.

# Ex1: no parameter



```
// a function, written in lambda expression/new style
() -> 98.6

// the same function, written in normal/old style
double f() {
    return 98.6;
}
```

# Ex2: with parameters



```
// a function, written in lambda expression/new style
(n) -> 1.0 / n

// the same function, written in normal/old style
double f(int n){
    return 1.0 / n;
}
```

# Ex3: omits the brackets



```
// a function, written in lambda expression/new style
n -> (n%2) == 0

// the same function, written in normal/old style
boolean f(int n){
    if(n % 2 == 0){
        return true;
    }
    else{
        return false;
    }
```

# Functional interfaces



- A functional interface is an interface that contains one and only one abstract method.

# Ex1



```java
package package11;

// written in lamda expression style
public class Main {
    public static void main(String[] args) {
        MyValue x;
        // Implement method MyValue.getValue()
        x = () -> 98.6;

        // Execute the lambda expression
        System.out.println(x.getValue());
    }
}

interface MyValue {
    double getValue();
}
```

# Ex1 (old style)



```
package package12;

// the same source code, written in normal style
public class Main {
    public static void main(String[] args) {
        MyValue x = new A();
        System.out.println(x.getValue());
    }
}

interface MyValue {
    double getValue();
}
class A implements MyValue{
    public double getValue(){
        return 98.6;
    }
}
```

# Ex2: with parameters



```
package package13;

public class Main {
    public static void main(String[] args) {
        MyParamValue x = (n) -> 1.0 / n;
        System.out.println(x.getValue(4.0));
    }
}

interface MyParamValue{
    double getValue(double v);
}
```

# Ex3: error cases



```
package package14;

public class Main {
    public static void main(String[] args) {
        MyParamValue x = (n) -> 1.0 / n;
        System.out.println(x.getValue(4.0));

        // error
        MyParamValue y = (n) -> "Three";
        System.out.println(y.getValue(3));

        // error
        MyParamValue z = () -> 3.14;
        System.out.println(z.getValue(5));
    }
}

interface MyParamValue{
    double getValue(double v);
}
```

# Practical example (1/2)



```
package package15;

// lamda expression style
public class LambdaWithMultiInstances {

    public static void main(String[] args) {
        MyComparator less = (a, b) -> {
            return a < b;
        };
        MyComparator equal = (a, b) -> {
            return a == b;
        };
        MyComparator greater = (a, b) -> {
            return a > b;
        };

        System.out.println("Less than? " + less.compare(3, 10));
        System.out.println("Equal to? " + equal.compare(3, 10));
        System.out.println("Greater than? " + greater.compare(3, 10));
    }
}
```

# Practical example (2/2)



```
package package15;

// lamda expression style
interface MyComparator {

    boolean compare(int a, int b);
}
```

# Practical example with normal classes



```
// normal style
class LessThan implements MyComparator{
    ...
}
class EqualTo implements MyComparator{
    ...
}
class GreaterThan implements MyComparator{
    ...
}
```

# Practical example with anonymous classes



```
// anonymous class style
MyComparator less = new MyComparator(){
...
}
MyComparator equal = new MyComparator(){
...
}
MyComparator greater = new MyComparator(){
...
}
```

# Block lambda expressions



```
package package16;

public class LambdaWithBlock {
    public static void main(String[] args) {
        MyInterface sum = (n) -> {
            int ans = 0;
            for (int i = 1; i <= n; i++) {
                ans += i;
            }
            return ans;
        };

        System.out.println(sum.func(10));
    }
}

interface MyInterface {

    int func(int n);
}
```

# Use lambda expression as an argument



```
package package17;

public class LambdaArgument {

    public static void main(String[] args) {
        MyComparator less = (a, b) -> {
            return a < b;
        };

        System.out.println("Less than? " + foo(3, 10, less));
    }

    public static boolean foo(int x, int y, MyComparator comp){
        return comp.compare(x, y);
    }
}

interface MyComparator {

    boolean compare(int a, int b);
}
```

# Lambda expressions and variable capture



```
package package18;

public class LambdaWithVariableCapture {
    public static void main(String []args){
        int num = 10;
        MyInterface x = (n) -> {
            int v =  num + n;
            //// an error use
            // num++;
            return v;
        };
        // an error use
        // num++;
        System.out.println(x.func(8));
    }
}

interface MyInterface{
    int func(int n);
}
```

# Lambda expression with exception



```
package package19;

public class LambdaWithExceptionDemo {
    public static void main(String[] args) {
        Reader reader = null;
        MyIOInterface x = (rdr) -> {
            rdr.read();
        };

        try {
            reader = new FileReader("test.txt");
            x.func(reader);
        } catch (FileNotFoundException ex) {
            System.out.println(ex.getMessage());
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}

interface MyIOInterface {
    void func(Reader reader) throws IOException;
}
```

# Method references



- A method reference provides a way to refer to a method without executing it.

# Method references to static methods (1/2)



```
package package20;

public class LambdaWithMethodReferencesStaticDemo {

    static boolean numTest(IntPredicate p, int v) {
        return p.test(v);
    }

    public static void main(String[] args) {
        System.out.println("10 is event? " + numTest(Utility::isEven, 10));
        System.out.println("11 is odd? " + numTest(Utility::isOdd, 11));
    }
}
```

# Method references to static methods (2/2)



```
package package20;

interface IntPredicate {

    boolean test(int n);
}

class Utility {

    static boolean isEven(int n) {
        return n % 2 == 0;
    }

    static boolean isOdd(int n) {
        return n % 2 != 0;
    }
}
```

# Method references to instance methods (1/2)



```
package package21;

public class LambdaWithMethodReferencesInstanceDemo {

    static boolean numTest(IntPredicate p, int v) {
        return p.test(v);
    }

    public static void main(String[] args) {
        Utility util = new Utility();

        System.out.println("10 is event? " + numTest(util::isEven, 10));
        System.out.println("11 is odd? " + numTest(util::isOdd, 11));
    }
}
```

# Method references to instance methods (2/2)



```
package package21;

interface IntPredicate {

    boolean test(int n);
}

class Utility {

    boolean isEven(int n) {
        return n % 2 == 0;
    }

    boolean isOdd(int n) {
        return n % 2 != 0;
    }
}
```

# Constructor references (1/2)



```
package package22;

public class LambdaWithMethodReferencesConstructorDemo {

    public static void main(String[] args) {
        MyInterface x = MyClass::new;

        MyClass c = x.func("Testing");
        System.out.println(c);
    }
}
```

# Constructor references (2/2)



```
package package22;

interface MyInterface {

    MyClass func(String s);
}

class MyClass {

    String str;

    MyClass() {
        str = "Empty";
    }

    MyClass(String s) {
        str = s;
    }

    @Override
    public String toString() {
        return str;
    }
}
```

# Predefined functional interfaces



```
UnaryOperator<T>
BinaryOperator<T>
Consumer<T>
Supplier<T>
Function<T, R>
Predicate<T>
```

# Predefined functional interfaces



```
package package23;

import java.util.function.BinaryOperator;

public class LambdaWithPredefinedDemo {

    public static void main(String[] args) {
        BinaryOperator<Integer> add = (a, b) -> a + b;
        System.out.println(add.apply(10, 20));
    }
}
```

# Predefined functional interfaces



```
package package24;

import java.util.ArrayList;

public class LambdaWithForEachDemo {

    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        numbers.add(5);
        numbers.add(9);
        numbers.add(8);
        numbers.add(1);

        numbers.forEach((x) -> {
            System.out.println(x);
        });
    }
}
```

# Reference



- https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
- https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html
- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- https://www.baeldung.com/java-8-lambda-expressions-tips

# Question ?