

A photograph of a large, two-story, light-colored building with a red-tiled roof and a central tower, surrounded by green grass and trees. The building is the Maharishi University of Management.

# MAHARISHI UNIVERSITY of MANAGEMENT

*Engaging the Managing Intelligence of Nature*

## Computer Science Department

**CS390 Fundamental Programming  
Practices (FPP)  
Professor Paul Corazza  
& A. McCollum**

# Lecture 6:

## Nested Classes

# Wholeness of the Lesson

Nested classes allow classes to play the roles of a variable: instance variable, static variable and local variable. This provides more expressive power to the Java language. Likewise, it is the hidden, unmanifest dynamics of consciousness that are responsible for the huge variety of expressions in the manifest world.

# Outline of Topics

1. **Definitions of Nested and Inner Class**
2. Four Types of Nested Classes: Member, Static, Local, and Anonymous
3. Using Lambda Expressions in Place of Anonymous Inner Classes – MPP topic
  - Syntax of lambda expressions
  - Lambdas as implementers of functional interfaces
4. Comparator, Another Functional Interface
  - Example: Typical Implementation
  - Example: Implementation using an anonymous class
  - Example: Implementation using a lambda expression

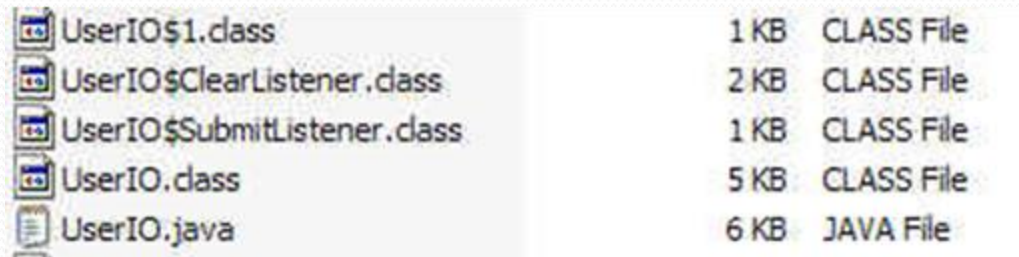


# Definition and Types of Nested Classes

- A class is a *nested class* if it is defined *inside* another class. (Note: This is different from having multiple classes defined in the same file.) A nested class is an *inner class* if it has access to all members (variables and methods) of its enclosing class (described below).
- Four kind of nested classes:
  - member
  - static
  - local
  - anonymous
- Of these, member, local, and anonymous are all called *inner* classes.
- An alternative, but equivalent, definition of *inner class* is “any non-static nested class”
- Sometimes in books you will see the term “static inner class” – this is simply loose language for static nested class.
- See demos for nested classes in package `lesson6_innerexamples`

# Definition and Types of Nested Classes

- The compiler adds code to nested class definitions and to the enclosing class to support the features of nested classes. When you compile the `UserIO` class (which has two member inner classes), you will see the inner classes explicitly named in `.class` files, using a '\$' in their names to distinguish them as nested classes.



UserIO\$1.class	1 KB	CLASS File
UserIO\$ClearListener.class	2 KB	CLASS File
UserIO\$SubmitListener.class	1 KB	CLASS File
UserIO.class	5 KB	CLASS File
UserIO.java	6 KB	JAVA File

Note: In `UserIO`, the `EventQueue.invokeLater` method defines an *anonymous* inner class – such a class is never given a name within the Java code. As the screen shot shows, the JVM handles this by naming such inner classes by number, starting with '1'.

- It is possible to make inner classes `private` and also `static` (see below) – these keywords cannot be used with ordinary classes.

# Main Point

Classes are the fundamental concept in Java – programs are built from classes. With nested classes, Java makes it possible for this fundamental construct to play the roles of instance variable (member inner classes), static variable (static nested classes), and local variable (local inner classes). Likewise, in the unfoldment of creation, pure consciousness, pure intelligence assumes the role of creative intelligence. In all creation we find pure intelligence in the guise of individual expressions, individual existences, assuming diversified roles.

# Outline of Topics

1. Definitions of Nested and Inner Class
2. **Four Types of Nested Classes: Member, Static, Local, and Anonymous**
3. Using Lambda Expressions in Place of Anonymous Inner Classes
  - Syntax of lambda expressions
  - Lambdas as implementers of functional interfaces
4. Comparator, Another Functional Interface
  - Example: Typical Implementation
  - Example: Implementation using an anonymous class
  - Example: Implementation using a lambda expression



# Member Inner Class Example

```
public class InnerMemberExample {  
    class StudentGrade {  
        String studentLetterGrade;  
        String studentId;  
  
        public String toString() {  
            return "Student with ID " + studentId  
                + " has grade " + studentLetterGrade;  
        }  
    }  
  
    StudentGrade stGrade = new StudentGrade();  
    {  
        stGrade.studentLetterGrade = "A";  
        stGrade.studentId = "123654";  
    }  
    void printGrade() {  
        System.out.println(stGrade);  
    }  
    public static void main(String[] args) {  
        new InnerMemberExample().printGrade();  
    }  
}
```

# Static Class Example

```
public class InnerMemberExample {  
    static class StudentGrade {  
        String studentLetterGrade;  
        String studentId;  
  
        public String toString() {  
            return "Student with ID " + studentId  
                + " has grade " + studentLetterGrade;  
        }  
    }  
  
    StudentGrade stGrade = new StudentGrade();  
    {  
        stGrade.studentLetterGrade = "A";  
        stGrade.studentId = "123654";  
    }  
    void printGrade() {  
        System.out.println(stGrade);  
    }  
    public static void main(String[] args) {  
        new InnerMemberExample().printGrade();  
    }  
}
```

# Local Class Example

local to a method

```
public class LocalClassExample {  
    void printGrade(String sGrade, String sGradeId) {  
        class StudentGrade {  
            String studentLetterGrade;  
            String studentId;  
  
            public String toString() {  
                return "Student with ID " + studentId  
                    + " has grade " + studentLetterGrade;  
            }  
        }  
        StudentGrade stGrade = new StudentGrade();  
        {  
            stGrade.studentLetterGrade = sGrade;  
            stGrade.studentId = sGradeId;  
        }  
        System.out.println(stGrade);  
    }  
    public static void main(String[] args) {  
        new LocalClassExample().printGrade("A", "123654");  
    }  
}
```

# Anonymous Class Example

```
public class AnonymousClassExample {  
    interface IGrade {  
        void printGradeInfo();  
    }  
    void printGrade(String sGrade, String sGradeId) {  
        (new IGrade() {  
            String studentLetterGrade = sGrade;  
            String studentId = sGradeId;  
  
            public String toString() {  
                return "Student with ID " + studentId  
                    + " has grade " + studentLetterGrade;  
            }  
            public void printGradeInfo() {  
                System.out.println(this);  
            }  
        }).printGradeInfo();  
    }  
  
    public static void main(String[] args) {  
        new AnonymousClassExample().printGrade("A", "123654");  
    }  
}
```

# Member Inner Classes

Begin with an example from Swing

```
public class MyFrame extends JFrame {  
    private JLabel label;  
    private JTextField text;  
    ...  
    class ButtonListener implements ActionListener{  
        public void actionPerformed(ActionEvent evt){  
            text.setText("button press");  
        }  
    }  
}
```



# Member Inner Class Syntax and Rules

- Member inner classes, like other members of the class, can be declared public or private, or may have package level access or may be *protected*. In the example, they have private level access.
- Member inner classes have access to all fields and methods of the enclosing class, including private fields and methods. No explicit reference to an enclosing class instance is needed.

In the example, notice how the `JTextField` is accessed.

- Likewise, the outer class can access private variables and methods in the inner class, but only with reference to an inner class instance that has already been created.

# Example

```
public class MyClass {  
    private String s = "hello";  
    public static void main(String[] args){  
        new MyClass();  
    }  
    MyClass() {  
        MyInnerClass myInner = new MyInnerClass();  
        System.out.println(myInner.intval);  
        myInner.innerMethod();  
    }  
    private class MyInnerClass {  
        private int intval = 3;  
        private void innerMethod(){  
            System.out.println(s);  
        }  
    }  
}
```

//Output:

```
3  
hello
```

- Like ordinary classes, when a member inner class is instantiated, it has an implicit parameter `'this'`. The `'this'` of the enclosing class is accessible from within the inner class. The `'this'` of the inner class is accessible from within itself, but *not* from the enclosing class.

# Example

```
Class MyOuterClass {
    MyInnerClass inner;
    private String param;
    MyOuterClass(String param) {
        inner = new MyInnerClass("innerStr");
        this.param = param; // the outer class version of this
    }
    void outerMethod() {
        System.out.println(inner.innerParam);
        inner.innerMethod();
        //String t = inner.this.innerParam; //compiler error
    }
    class MyInnerClass{
        private String innerParam;
        MyInnerClass(String innerParam) {
            //the inner class version of 'this'
            this.innerParam = innerParam;
        }
        void innerMethod() {
            //accessing enclosing class's version of this
            System.out.println(MyOuterClass.this.param);
            //same as the following
            System.out.println(param);
        }
    }
}

public static void main(String[] args) {
    (new MyOuterClass("outerStr")).outerMethod();
}
```

```
//OUTPUT:
innerStr
outerSt
outerStr
```

- To access the methods and variables of a member inner class, it is necessary to explicitly instantiate it – it is *not* instantiated automatically when the enclosing class is instantiated.

```
public class MyClass {  
    private String s = "hello";  
    MyInnerClass inner;  
    public static void main(String[] args){  
        new MyClass();  
    }  
    MyClass() {  
        System.out.println(inner.anInt); //NullPointerException  
        inner = new MyInnerClass();  
        System.out.println(inner.anInt); //OK  
    }  
    class MyInnerClass{  
        private int anInt = 3;  
        void innerMethod(){  
            System.out.println(s);  
        }  
    }  
}
```

- In a member inner class, no variable or method may be declared static. (By contrast, static members are allowed in a static nested class.)



- If the member inner class is sufficiently accessible (i.e., not private), it can be instantiated by a class other than the enclosing class, as long as an instance of the enclosing class has already been created.

## Example:

```
class ClassA {
    class InnerClassA {
    }
}

class ClassB {
    ClassB() {
        ClassA a = new ClassA();
        ClassA.InnerClassA innerA = a.new InnerClassA(); //ok
    }
}

class ClassC {
    ClassC() {
        ClassA.InnerClassA innerA =
            new ClassA.InnerClassA(); //illegal, 'new' requires an
                                   //enclosing instance
    }
}
```

- **Best Practice.** A member inner class is typically a small specialized “assistant” that is exclusively owned by its enclosing class. Often used in a GUI class to enclose event handlers (though listeners can be handled in other ways too). Consequently, it is not good practice to access a member inner class from outside the enclosing class, since this undermines the overall purpose of this type of nested class.

# Exercise 6.1

In each class, what happens? Is there a compiler error? If not, what happens when the main method is run? Runtime error? Successful execution? If successful, what is printed to the console?

```
public class MyClass {
    private MyInner inner;
    public MyInner getMyInner() {
        return inner;
    }
    private class MyInner {
        private int innerInt;
        MyInner(int x) {
            innerInt = x;
        }
    }
    public static void main(String[] args) {
        MyClass mc = new MyClass();
        MyInner mi = mc.getMyInner();
        System.out.println(mi.innerInt);
    }
}
```

```
public class Number {
    public static final Number ONE = newOne();
    public static final Number TWO = newTwo();

    private class One extends Number {
        @Override
        public String toString() {
            return "one";
        }
    }
    private Number() {
        //do nothing
    }
    private class Two extends Number {
        @Override
        public String toString() {
            return "two";
        }
    }
    private static Number newOne() {
        return (new Number()).new One();
    }
    private static Number newTwo() {
        return (new Number()).new Two();
    }
    public static void main(String[] args) {
        System.out.println(Number.ONE);
        System.out.println(Number.TWO);
    }
}
```

# Main Point

Inner classes – a special kind of nested class – have access to the private members of their enclosing class. The most commonly used kind of inner class is a *member* inner class. Likewise, when individual awareness is awake to its fully expanded, self-referral state, the memory of its ultimate nature becomes lively.

# Static Nested Classes

- If a nested class is defined using the `static` keyword, it becomes a static nested class.
- Static nested classes do not have access to instance variables and methods of the enclosing class. A static nested class is in effect a top level class that has been “packaged” differently; it has the same access to the enclosing class variables and methods as another class located in the same package.

```
public class Main {  
    public int i = 4;  
    public int getInt() {  
        return 3;  
    }  
    static class NestedClass {  
        public void innerMethod() {  
            int j = i;           //compiler error  
            int k = getInt();    //compiler error  
        }  
    }  
}
```



- It is possible to declare static variables and methods in a static nested class; however, static nested classes may also have instance variables and methods.
- As with member inner classes, the enclosing class of a static nested class has access to the nested class's private variables and methods, with reference to an instance.
- Unless a static nested class is declared private, other classes can instantiate it, but the syntax is different from that for member inner classes. For example:

```
MyClass.MyStaticNestedClass cl =  
    new MyClass.MyStaticNestedClass();
```

- A static nested class may not be defined inside an instance inner class. (Recall that static variables and methods cannot be declared inside a member inner class either.)

```

public class MyClass {
    private String s = "hello";
    public static void main(String[] args){
        new MyClass();
    }
    MyClass() {
        //access static methods in the usual way
        MyStaticNestedClass.myStaticMethod();

        //access instance methods in the usual way too
        //except that now private methods are also accessible
        MyStaticNestedClass cl = new MyStaticNestedClass();
        cl.myOtherMethod();

        //as with inner classes, private instance vbles are accessible
        int y = cl.x;
    }
    static class MyStaticNestedClass {
        private int x = 0;
        static void myStaticMethod() {
            String t = s; //compiler error -- no access
        }
        private void myOtherMethod() {
        }
    }
}

class AnotherClass {
    public static void main(String[] args){
        MyClass.MyStaticNestedClass cl = new MyClass.MyStaticNestedClass(); //OK
        MyClass m = new MyClass();

        //the following is illegal-- compiler error

        MyClass.MyStaticNestedClass cl2 = m.new MyStaticNestedClass();
    }
}

```

- **Best Practice.** Static nested classes should be thought of as ordinary (“top-level” or “first class”) classes that are “privately packaged”. Usually not accessed from outside, but it’s not necessarily bad practice to do so since static nested classes are “top level” classes.
  - The book gives an example of a `Pair` class that is defined within another class `ArrayAlg`; could make `Pair` an ordinary (externally defined) class, but making it static nested class controls the namespace – if another `Pair` class exists in the application, there will be no conflict.
  - In the Java API, `LinkedList.Entry` and `HashMap.Entry` are examples of static nested classes.

# Nested Classes: Static And Non-static

- A nested static class doesn't need to use a reference to Outer class, but the non-static nested classes or Inner classes require an Outer class reference.
- Inner class (any of the non-static nested classes) can access both static and non-static members of Outer class. A static class cannot access non-static members of the Outer class. It can access only static members of Outer class.
- An instance of Inner class cannot be created without an instance of an Outer class. An Inner class can reference data and methods in the Outer class in where it nests, so we don't need to pass a reference of an object to the constructor of the Inner class.

# Local Inner Classes

- Local inner classes are defined entirely within the body of a method.
- Access specifiers (public, private, etc) are not used to affect access of the inner class since access to the inner class is always restricted to the local access within the method body. (However, methods in the inner class may be – and may need to be – given some access specifier – see the example.)
- Local inner classes have access to instance variables and methods in the enclosing class; they also have access to *local variables* – variables inside the method body, as well as parameters passed in to the method – as long as these local variables are *effectively final* (this means that your code cannot change the values of these variables during execution).

[Note: Prior to jdk 1.8, such variables had to be declared **final**. Examples are shown later in these slides.]



# Example: Implementing an ActionListener As a Member Inner Class

```
public class MyFrameInner extends JFrame {
    //make the text field and label instance variables in MyFrame
    JLabel label;
    JTextField text;
    public MyFrameInner() {
        initializeWindow();
        JPanel mainPanel = new JPanel();
        text = new JTextField(10);
        label = new JLabel("My Text");
        JButton button = new JButton("My Button");

        //Registering the listener with button using a member inner class
        button.addActionListener(new ButtonListener());
        mainPanel.add(text);
        mainPanel.add(label);
        mainPanel.add(button);
        getContentPane().add(mainPanel);
    }
    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            JButton butn = (JButton)evt.getSource();
            System.out.println("Button height = " + butn.getSize().height);
            text.setText("button press");
        }
    }
}
```

...

# Example: Implementing an ActionListener As a Local Inner Class

```
public class MyFrameLocal extends JFrame {
    JLabel label;
    JTextField text;
    public MyFrameLocal() {
        initializeWindow();
        JPanel mainPanel = new JPanel();
        text = new JTextField(10);
        label = new JLabel("My Text");
        JButton button = new JButton("My Button");

        //Registering an ActionListener with button using a local inner class
        addActionListener(button);
        mainPanel.add(text);
        mainPanel.add(label);
        mainPanel.add(button);
        getContentPane().add(mainPanel);
    }

    //Using a local inner class inside a custom method for adding a
    //listener to a button
    private void addActionListener(final JButton b) {
        class ButtonListener implements ActionListener {
            public void actionPerformed(ActionEvent evt) {
                System.out.println("Button height = " + b.getSize().height);
                text.setText("button press");
            }
        }
        b.addActionListener(new ButtonListener());
    }
}
```

# Exercise 6.2

Does the following code compile? If so, what happens when the main method is run? Is there a runtime error? If not, what is printed to the console?

```
public class Outer {  
    private int data = 10;  
    public static void main(String[] args) {  
        Outer outer = new Outer();  
        outer.printVal(20);  
    }  
    void printVal(int bound) {  
        if(data < bound) {  
            class Inner {  
                public int getValue() {  
                    return data;  
                }  
            }  
            Inner inner = new Inner();  
            System.out.println("Inside inner: " + inner.getValue());  
        } else {  
            System.out.println("Inside outer: " + (data - bound));  
        }  
    }  
}
```

# Advantages of Local Inner Classes

Two reasons for preferring local inner classes to member inner classes:

1. A local inner class is defined precisely where it is needed. This makes it easier to maintain, and makes code easier to follow
2. A local inner class can be used only for one purpose – the purpose of its enclosing method. Therefore, local inner classes are used to provide *strong encapsulation*.

In the example, there is no reason to make the `ButtonListener` accessible to any method (in any class) other than the method that attaches the listener to the button

# Issues Concerning Local Inner Classes

- *Artificial Auxiliary Method.* Sometimes the desired functionality of the inner class is not naturally associated with a method, so a method has to be artificially created in order to specify a local inner class. This is what happened in the example – an artificial `addActionListener` method was introduced to permit the definition of a local inner class.
- *Local Inner Classes Should Be Small.* When the inner class requires more than a small amount of code, it should not be squeezed inside the body of a method – the method body would become too big, harder to read, and maintenance of the method becomes more difficult. In such cases, member inner classes are preferable.
- *Avoid Including Local Inner Classes Involved in a Loop.* If the method in which the local inner class is defined is called from a loop, the inner class will be instantiated repeatedly. This behavior can be costly – in such cases, it is usually better to move the class out as a member inner class (or even a top-level class) and instantiate it just once; if its values need to take on different values as a loop executes, these can be set using setter methods.

# Anonymous Inner Classes

- An anonymous inner class is a kind of inner class that is defined – without a name – and instantiated in a single block of code.
- Sometimes an anonymous inner class is defined – like local inner classes – within a method body. In that case, the code is even more compact, and has the same advantages as a local inner class.
- In general, an anonymous inner class is used to create an "on the fly" subclass of a known class or an "on the fly" implementation of a known interface.

# Example: Implementing an ActionListener with an Anonymous Inner Class

```
public class MyFrameAnonymous extends JFrame {
    JLabel label;
    JTextField text;
    public MyFrameAnonymous() {
        initializeWindow();
        JPanel mainPanel = new JPanel();

        text = new JTextField(10);
        label = new JLabel("My Text");
        final JButton button = new JButton("My Button");

        //Registering an ActionListener with button as an anonymous inner class
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                System.out.println("Button height = " + button.getSize().height);
                text.setText("button press");
            }
        });
        mainPanel.add(text);
        mainPanel.add(label);
        mainPanel.add(button);
        getContentPane().add(mainPanel);
    }
    ...
}
```



# Advantages to Anonymous Inner Classes

- They can be used in place of local inner classes without creating an artificial auxiliary method (as in the previous slide)
- They provide the same strong encapsulation as local inner classes.
- They have wider applicability than local inner classes: Whenever either a subclass of a class or an implementation of an interface is needed, anonymous inner classes can be used – no enclosing method is necessary.

# Disadvantages to Anonymous Inner Classes

- Explicit constructors cannot be used within an anonymous inner class (constructors give a name to a class and anonymous inner classes have no name)
- The syntax can be confusing – hard to read and maintain.

# Outline of Topics

1. Definitions of Nested and Inner Class
2. Four Types of Nested Classes: Member, Static, Local, and Anonymous
3. **Using Lambda Expressions in Place of Anonymous Inner Classes**
  - **Syntax of lambda expressions**
  - **Lambdas as implementers of functional interfaces**
4. Comparator, Another Functional Interface
  - Example: Typical Implementation
  - Example: Implementation using an anonymous class
  - Example: Implementation using a lambda expression

# Lambda Expression (in JSE8+) Overview

## Discussed in MPP

Lambda expressions were added in Java 8.

- They allow developers to use functions as a method argument, or code as data.
- A function can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object, and can also be executed on demand.

# Lambda Expressions Syntax

Lambda expressions are like functions. They accept parameters as do functions.

Syntax: lambda operator -> body  
where lambda operator can be:

Zero parameter: () -> `System.out.println("Zero parameter lambda");`

One parameter:- (p) -> `System.out.println("One parameter: " + p);`

Multiple parameters :

(p1, p2) -> `System.out.println("Multiple parameters: " + p1 + ", " + p2);`

No need to use parentheses when the variable type can be inferred from context

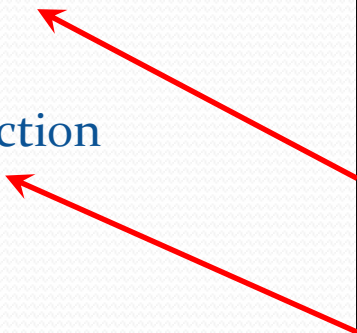
# Lambda Expressions Example

// Demo lambda expressions - implement a user defined functional interface.  
// A sample functional interface (An interface with single abstract method)

```
interface FuncInterface
{
    // An abstract function
    void abstractFunc(int x);

    // A non-abstract (or default) function
    //default void normalFunc()
    //{
    //    System.out.println("Hello");
    //}
}
```

The FuncInterface is a functional interface because it has on function that is not implemented “abstractFunc”. This is still a functional interface regardless of whether the “normalFunc” method is present or absent, because “normalFunc” is not abstract.



# Lambda Expressions Example (continued)

```
class Test
{
    public static void main(String args[])
    {
        // lambda expression to implement functional interface.
        // This interface by default implements abstractFunc()
        FuncInterface fobj = (int x)->System.out.println(2*x);

        // This calls above lambda expression and prints 10.
        fobj.abstractFunc(5);
    }
}
```



# Example based on Swing: An ActionListener with a Lambda Expression (for JSE8 and later)

```

JButton button = new JButton("My Button");
// 1st format
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        System.out.println(text.getText());
    }
});
// 2nd format, with same effect
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
    evt -> {
        System.out.println(text.getText());
    }
});
// 3rd format, with same effect
button.addActionListener(
    evt -> System.out.println(text.getText());
)
```

← using anonymous inner class

← These are two ways that the above Anonymous Inner Class can be expressed using a lambda expression  
←

# Example: An ActionListener with a Lambda Expression (Repeat of previous slide)

```
JButton button = new JButton("My Button");

button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        System.out.println(text.getText());
    }
});

button.addActionListener(
    evt -> {
        System.out.println(text.getText());
    }
);

button.addActionListener(
    evt -> System.out.println(text.getText());
);
```

using anonymous  
inner class



using a lambda  
expression



# About the Code Sample

- A lambda expression has been used to replace an implementation of the `ActionListener` interface.
- Note: `ActionListener` has just one (abstract) method `actionPerformed`, so it is a functional interface. The `actionPerformed` method takes one argument of type `ActionEvent` and maps it to a block of code, representing the action to be performed.

The lambda expression just maps the `ActionEvent evt` to a block of code

```
evt -> System.out.println(text.getText());
```

- Using curly braces `{, }` to mark off the block is optional when there is just one line but required if there is more than one line

# (continued)

```
button.addActionListener(  
    evt -> System.out.println(text.getText());  
)
```

*How is the compiler able to make sense of this expression?*

Since the `addActionListener` method takes an `ActionListener` argument, which has only one declared method (`actionPerformed`), and since `actionPerformed` takes just one argument (`ActionEvent`), the compiler infers:

1. "evt" is of type `ActionEvent`
2. the code on the right side of the arrow must represent an implementation of the `actionPerformed` method

This logic – which determines the types involved – is called *target typing*.

# Outline of Topics

1. Definitions of Nested and Inner Class
2. Four Types of Nested Classes: Member, Static, Local, and Anonymous
3. Using Lambda Expressions in Place of Anonymous Inner Classes
  - Syntax of lambda expressions
  - Lambdas as implementers of functional interfaces
4. **Comparator, Another Functional Interface**
  - **Example: Typical Implementation**
  - Example: Implementation using an anonymous class
  - Example: Implementation using a lambda expression

# The Comparator Interface

**Problem:** Suppose you have an array `arr` of `Person` objects and you want to sort the array in this way:

```
Arrays.sort(arr)
```

The compiler will complain because there is no natural ordering on `Person`, and `Person` does not (in this case) implement `Comparable`.

**Solution:** Implement a `Comparator` for `Person`.  
(Another solution would be to have `Person` implement `Comparable`.)

[See `lesson6.comparators` for several demos.]

- Declare person1 to be "less than" person2 if the name of person1 comes before the name of person2 :

```
public class NameComparator implements Comparator<Person>{  
  
    @Override  
    public int compare(Person p1, Person p2) {  
        return p1.getName().compareTo(p2.getName());  
    }  
}
```

- Then you can sort an array of Persons if you also pass in an instance of the NameComparator:

```
public static void main(String[] args) {  
    Person[] persons = PersonData.personData;  
    Arrays.sort(persons, new NameComparator());  
    System.out.println(Arrays.toString(persons));  
}
```



- NameComparator can also be defined as a lambda expression since it is an implementation of a functional interface:

```
Person[] persons = PersonData.personData;  
Comparator<Person> comp  
    = (p1,p2)-> p1.getName().compareTo(p2.getName());  
Arrays.sort(persons, comp);  
System.out.println(Arrays.toString(persons));
```

- We can implement the NameComparator instead as an anonymous inner class, created dynamically within the sort method

```
public static void main(String[] args) {  
    Person[] persons = PersonData.personData;  
    Arrays.sort(persons, new Comparator<Person>() {  
        public int compare(Person p1, Person p2) {  
            return p1.getName().compareTo(p2.getName());  
        }  
    });  
    System.out.println(Arrays.toString(persons));  
}
```

- Instead of a named lambda expression, we can define an anonymous lambda expression as one of the arguments in the `sort` method.

Before:

```
public static void main(String[] args) {  
    Person[] persons = PersonData.personData;  
    Arrays.sort(persons, new Comparator<Person>() {  
        public int compare(Person p1, Person p2) {  
            return p1.getName().compareTo(p2.getName());  
        }  
    });  
    System.out.println(Arrays.toString(persons));  
}
```

After:

```
public class MainLambdaAnonymous {  
    public static void main(String[] args) {  
        Person[] persons = PersonData.personData;  
        Arrays.sort(persons,  
            (p1,p2)-> p1.getName().compareTo(p2.getName()));  
        System.out.println(Arrays.toString(persons));  
    }  
}
```

# Exercise 6.3

Refactor the code below so that the instance of the inner class is replaced by a *named* lambda expression. Notice that `ValGetter` is a functional interface. *Hint*: Look at how `NameComparator` was replaced by a lambda.

```
public interface ValGetter {  
    int getValue();  
}
```

```
public class Outer {  
    private int data = 10;  
    public static void main(String[] args) {  
        Outer outer = new Outer();  
        outer.printVal(20);  
    }  
    void printVal(int bound) {  
        if(data < bound) {  
            //Replace definition of Inner and call to inner.getValue()  
            //with a lambda expression.  
            class Inner implements ValGetter {  
                public int getValue() {  
                    return data;  
                }  
            }  
            Inner inner = new Inner();  
            System.out.println("Inside inner: " + inner.getValue());  
        } else {  
            System.out.println("Inside outer: " + (data - bound));  
        }  
    }  
}
```

# Exercise 6.3 - Solution

Refactor the code below so that the instance of the inner class is replaced by a *named* lambda expression. Notice that `ValGetter` is a functional interface. *Hint*: Look at how `NameComparator` was replaced by a lambda.

```
public class Outer {
    private int data = 10;
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.printVal(20);
    }
    void printVal(int bound) {
        if(data < bound) {
            ValGetter vg = () -> data;
            System.out.println("Using lambda: " + vg.getValue());

        } else {
            System.out.println("Inside outer: " + (data - bound));
        }
    }
}
```

```
public interface ValGetter {
    int getValue();
}
```

# Summary

Java has four kinds of nested classes: member, static, local and anonymous

- *Member inner classes* are used as private support within a class, much as instance variables and private methods are used. They have full access to the instance variables and methods of the enclosing class.
- *Static nested classes* are top-level classes that are naturally associated with their enclosing class, but have no special access to the data or behavior of the enclosing class.
- *Local inner classes* are defined entirely within a method body; *anonymous inner classes* make it possible to define a class at the moment that an instance of the class is created. Both types of inner classes are accessible only within the local context in which they are defined, resulting in an extreme form of encapsulation.
- When inner classes are used as implementers of *functional interfaces* – like `ActionListener` or `Comparator` – they can be replaced by *lambda expressions*, which extract from the inner class implementation the bare functional essence, resulting in more compact and easier to understand code.

# Connecting the Parts of Knowledge With the Wholeness of Knowledge

*Inner classes retain the memory of their "unbounded" context*

1. A *nested class* is a class that is defined inside another class.
  2. An *inner class* is a nested class that has full access to its context, its enclosing class.
- 
3. **Transcendental Consciousness:** TC is the unbounded context for individual awareness.
  4. **Wholeness moving within itself:** When individual awareness is permanently and fully established in its transcendental "context" – pure consciousness – every impulse of creation is seen to be an impulse of one's own awareness.

