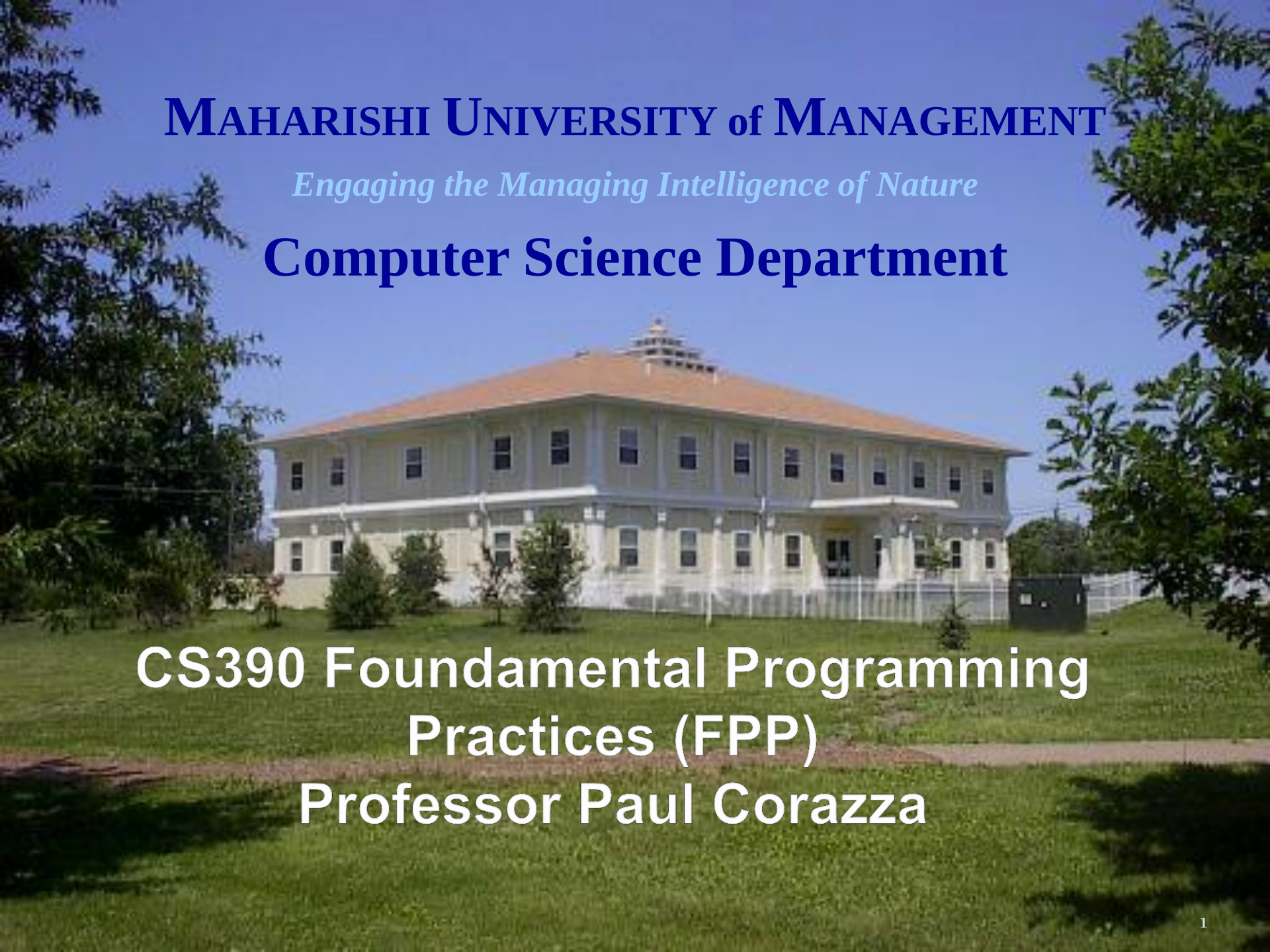

MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS390 Foundamental Programming Practices (FPP)
Professor Paul Corazza

# Lecture 6:
# Nested Classes

# Wholeness of the Lesson

Nested classes allow classes to play the roles of instance variable, static variable and local variable, providing 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

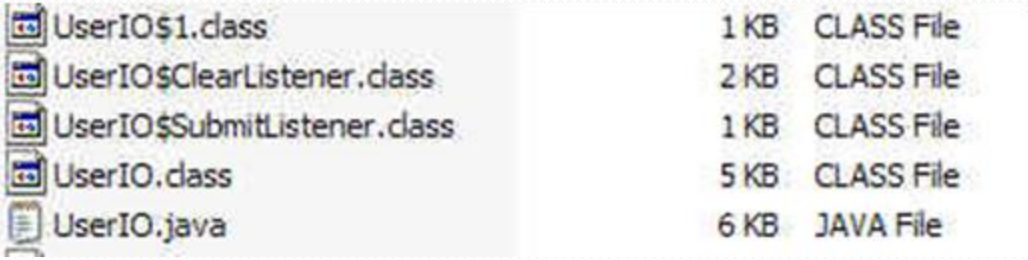
1. Definitions of Nested and Inner Class
2. Four Types of Nested Classes: Member, Static, Local, and Anonymous
3. Using Lambda Expressions in Place of Anonymous Inner Classes
   - Syntax of lambda expressions
   - Lambdas as implementers of functional interfaces
4. Comparator, Another Functional Interface
   - Example: Implementation using a local inner class
   - Example: Implementation using a lambda expression
5. Application of Nested Classes: Implementing the Singleton Pattern

# 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.




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 intelligence assumes the role of creative intelligence – in all of creation we find pure intelligence in the guise of individual expressions, individual existences, assuming diversified roles.

# Member Inner Classes

Begin with an example from the `UserIO` class:

```
public class UserIO extends JFrame  {
    private JTextArea upperText;
    private JTextArea lowerText;
    public UserIO() {
          initializeWindow();
          defineMainPanel();
          getContentPane().add(mainPanel);
    }
    private void initializeWindow() {
          //. . . //

    }
```

```
    // . . . //
    private void defineMiddlePanel() {
         middleSubpanel = new JPanel();
         JPanel textPanel = new JPanel();
         JPanel buttonPanel = new JPanel();

         //define textPanel
         middleSubpanel.add(textPanel,BorderLayout.NORTH);

         //define button panel
         buttonPanel = new JPanel();
         JButton submitButn = new JButton(SUBMIT);
         submitButn.addActionListener(new SubmitListener());
         buttonPanel.add(submitButn);
         middleSubpanel.add(buttonPanel,BorderLayout.SOUTH);

    }
    class SubmitListener implements ActionListener {
         public void actionPerformed(ActionEvent evt) {
                 inputString = upperText.getText();
         }
    }
    class ClearListener implements ActionListener {
         public void actionPerformed(ActionEvent evt) {
             lowerText.setText("");
             System.out.println("Clearing output text area.");
         }
    }
}
```

# 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* – discussed in Lesson 7). In the example, they have package 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 `JTextAreas` are 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.

# 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.

# 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());
    }

}
```

# 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 to 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.
- See package `lesson6_moreanonymous` for more examples.

# 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.

# Example: Implementing an ActionListener with a Lambda Expression (for jse8 and later)

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

//Registering an ActionListener with button
//using lambda notation - requires jse8.
//Java figures out the type of evt using
//"target typing"
button.addActionListener(
    evt -> {
          System.out.println("Button height = " +
             button.getSize().height);
          text.setText("button press");
    }
);
```

# 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`. The `actionPerformed` method takes one argument of type `ActionEvent` and maps it to a block of code, representing the action to be performed.

  Recall: Associated with the argument `ActionEvent evt` was the block of code

```
{
  System.out.println("Button height = " +
      button.getSize().height);
  text.setText("button press");
}
```

- A lambda expression abstracts from the inner class syntax just this functional relationship:

```
evt -> {
      System.out.println("Button height = " +
         button.getSize().height);
      text.setText("button press");
}
```

# Notes about Lambda Expressions

- An interface that contains just one (abstract) method is called a *functional interface*
- An implementation of a functional interface is called a *functor*
- A *closure* is a functor that remembers the state of its enclosing environment
- Example: `ActionListener`
  - `ActionListener` is an interface with just one method – `actionPerformed` – so it is a *functional interface*
  - An anonymous inner class implementation of `ActionListener` is a *functor*
  - An anonymous inner class implementation of `ActionListener` is a *closure*, though local variables that are not *effectively final* are not accessible
  - A lambda expression can be used in place of an anonymous inner class

- The Java runtime determines the type of the function argument `evt` by context – a mechanism that is called *target typing*. Note: target typing relies on the fact that lambda expressions are used only with *functional interfaces*; there is just one possible function that is to be implemented, so the JVM can use this fact to help the determine the correct typing.

- Examples of lambda expressions:
  - `(a,b) -> a + b`
  - `() -> 5`
  - `() -> {System.out.println("Inside a block.");}`
  - `evt -> {text.setText("button press");}`

- Each of these expressions is a realization of some functional interface; determining *which* functional interface is realized in each case requires a deeper study (this is taken up in MPP).

# Comparator : Another Functional Interface

- The `Comparator` interface has just one abstract method, `compare`, so it is also a functional interface. It is used to define an order relationship for objects that don't have a natural ordering. For instance, numbers and strings have a natural ordering, so we can sort them. But how would you sort Employee objects?
- In practice, we may want to sort business objects in different ways. An `Employee` list could be sorted by name, salary or hire date.

| **Employee Data** | | |
|---|---|---|
| Name | Hire Date | Salary |
| Joe Smith | 11/23/2000 | 50000 |
| Susan Randolph | 2/14/2002 | 60000 |
| Ronald Richards | 1/1/2005 | 70000 |

- To accomplish this, you specify your own ordering on a class using the **Comparator** interface, whose only method is **compare()**.

- Like lists, in j2se5.0, `Comparators` are parametrized: You must specify a type for `T` in `Comparator<T>`. (See the examples.)

- The `compare()` method is expected to behave in the following way (so it can be used in conjunction with the `Collections` API):

  For objects `a` and `b`,

  - `compare(a,b)` returns a negative number if `a` is "less than" `b`
  - `compare(a,b)` returns a positive number if `a` is "greater than" `b`
  - `compare(a,b)` returns `0` if `a` "equals" `b`

- Example of a `Comparator` for the Employee class:

```
public class NameComparator implements Comparator<Employee> {
   @Override
   public int compare(Employee e1, Employee e2) {
     return e1.getName().compareTo(e2.getName());
   }
}
```

This `compare` method declares an ordering on `Employee`s so th`e2, e1` "comes befor betically before the nam

```
public static void main(String[] args) {
    Employee[] emps = {new Employee("Bob", 200000),
            new Employee("Anne", 150000),
            new Employee("Steve", 155000)};

    Arrays.sort(emps, new NameComparator());
    System.out.println(Arrays.toString(emps));

}
```

- Implementation using a lambda expression

```
public static void main(String[] args) {
    Employee[] emps = {new Employee("Bob", 200000),
            new Employee("Anne", 150000),
            new Employee("Steve", 155000)};

    Arrays.sort(emps, new NameComparator());
    System.out.println(Arrays.toString(emps));

    //using lambdas
    Arrays.sort(emps, (e1, e2) -> e1.getName().compareTo(e2.getName()));
    System.out.println(Arrays.toString(emps));

}
```

NOTE: `Comparators` will be discussed in more detail in Lesson 8. See package `lesson6_moreanonymous` for an implementation using anonymous classes.

# Local Variables and Local Inner Classes

Implementing a Comparator as a local inner class

```
public class EmployeeInfo {
    static enum SortMethod {BYNAME, BYSALARY};
    private boolean ignoreCase = true;
    public void setIgnoreCase(boolean b) {
        ignoreCase = b;
    }
    public void sort(List<Employee> emps, final SortMethod method) {
        class EmployeeComparator implements Comparator<Employee> {
            @Override
            public int compare(Employee e1, Employee e2) {
                    //local variable method must be final in java 7,
                    //effectively final in java 8
                    if(method == SortMethod.BYNAME) {
                        //instance vble ignoreCase does not need to be final
                        if(ignoreCase) return e1.name.compareToIgnoreCase(e2.name);
                        else return e1.name.compareTo(e2.name);
                    } else {
                        if(e1.salary == e2.salary) return 0;
                        else if(e1.salary < e2.salary) return -1;
                        else return 1;
                    }
            }
        }
        Collections.sort(emps, new EmployeeComparator());
    }
}
```

Compare method must be declared public to override Comparator method

Local and anonymous inner classes have access to instance variables of the enclosing class, but may use local variables only if they are *final*.

# Local Variables and Local Inner Classes

Implementing a Comparator with a lambda expression

```
public class EmployeeInfo {
    static enum SortMethod {BYNAME, BYSALARY};
    private boolean ignoreCase = true;
    public void setIgnoreCase(boolean b) {
        ignoreCase = b;
    }
    public void sort(List<Employee> emps, SortMethod method) {
        Collections.sort(emps, (e1,e2) ->
        {
                //local variable method must be effectively final,
                //but not necessarily final
                if(method == SortMethod.BYNAME) {
                    //instance vble ignoreCase does not need to be effectively final
                    //but should not be modified either
                    if(ignoreCase) return e1.name.compareToIgnoreCase(e2.name);
                    else return e1.name.compareTo(e2.name);
                } else {
                    if(e1.salary == e2.salary) return 0;
                    else if(e1.salary < e2.salary) return -1;
                    else return 1;
                }
        });
    }
```

Lambda expressions have access to instance variables of the enclosing class, but may use local variables only if they are *effectively final* – this means that the value of the variable never changes (this is compiler-checked). As of Java SE 8, this is also the rule for local and anonymous inner classes.

# Application of Nested Classes: Implementing the Singleton Pattern

There are several ways to implement the singleton pattern:

- Use lazy initialization to instantiate a private static instance variable. (Not threadsafe.)
- Store instance as a public static constant, constructed when class is loaded.
- Implement as an Enum. Also constructed when enum is loaded.
- Use Spring's Singleton Holder pattern, whereby the singleton is stored as a static variable of a static nested class. Permits lazy initialization and is threadsafe.

# 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.
- An important application of nested classes occurs in implementations of the *Singleton Pattern*.

# 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.