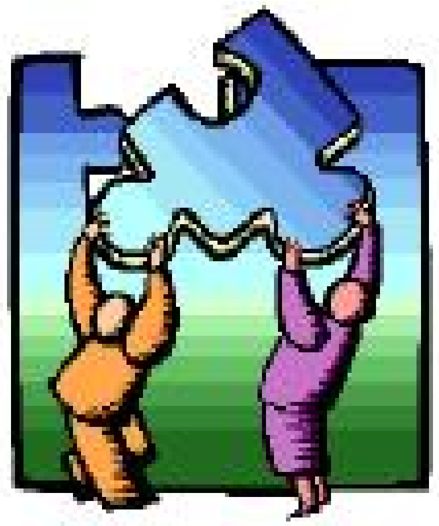
Java Nested and Inner Classes



Java Inner Classes

- Inner, or Nested, classes are standard classes declared within the scope of a standard top-level class
  - as a member just as fields/methods
  - inside a method (a.k.a. local/anonymous classes)
- Direct access to members of its enclosing definitions
  - Programming style similar to nested functions
- Extensively used in event-driven programming (e.g., AWT)

Complication (1): Scoping Rules

- Directly accessible members are in
  - ◆superclasses
  - ◆enclosing classes
  - ◆even superclasses of enclosing classes

```
class A {
  Object f; …
}
class B {
  Object f;
  class C extends A {
    void m() {
      … f … // which f?
}}}
```

Complication (2): Inheritance

- Almost any form of inheritance is allowed
  - ◆Inner class is inherited outside of the defined scope

```
class A { …
  class B { … }
}
class C extends A.B { … }
```

  - ◆Inner class can extend its enclosing class

```
class A { …
  class B extends A { … }
}
```

Kinds of Inner Classes

- There are different kinds of inner class and became available with Java 1.1.
  - A) Nested top-level or Static Member class
  - B) Member class
  - C) Local class
  - D) Anonymous class

A/ Nested Top-Level Classes

```
class outer {
  private static class NestedTopLevel {
    normal class stuff
  }
  normal class stuff
}
```

- Nested top-level classes are declared static within a top-level class (sort of like a class member)
- They follow the same rules as standard classes
  - `private static` classes cannot be seen outside the enclosing class
  - `public static` allows the class to be seen outside

LinkedStack

```
public class LinkedStack {
  private StackNode tos = null;

  private static class StackNode {
    private Object data; private StackNode next, prev;

    public StackNode( Object o ) { this( o, null ); }
    public StackNode( Object o, StackNode n ) {
      data = o; next = n;
    }
    public StackNode getNext() { return next; }
    public Object getData() { return data; }
  }
  public boolean isEmpty() { return tos == null; }
  public boolean isFull() { return false; }
  public void push( Object o ) { tos = new StackNode( o, tos ); }
  public void pop() { tos = tos.getNext(); }
  public Object top() { return tos.getData(); }
}
```

B/ Member Classes

- A member class is a nested top-level class that is not declared static
- This means the member class has a `this` reference which refers to the enclosing class object
- Member classes cannot declare static variables, methods or nested top-level classes
- Member objects are used to create data structures that need to know about the object they are contained in

Example

```
class Test {

  private class Member {
   public void test() {
     i = i + 10;
     System.out.println( i );
     System.out.println( s );
    }
  }
  public void test() {
    Member n = new Member();
    n.test();
  }
  private int i = 10;
  private String s = “Hello”;
}
```

`this` Revisited

- To support member classes several extra kinds of expressions are provided
  - `x = this.dataMember` is valid only if `dataMember` is an instance variable declared by the member class, not if `dataMember` belongs to the enclosing class
  - `x = EnclosingClass.this.dataMember` allows access to `dataMember` that belongs to the enclosing class
- Inner classes can be nested to any depth and the `this` mechanism can be used with nesting

this and Member Classes

```
public class EnclosingClass {
 private int i,j;

 private class MemberClass {
  private int i;  public int j;

  public void aMethod( int i ) {
   int a = i;                      // Assign param to a
   int b = this.i;                 // Assign member's i to b
   int c = EnclosingClass.this.i;  // Assign top-level's i to c
   int d = j;                      // Assign member's j to d
  }  }

  public void aMethod() {
   MemberClass mem = new MemberClass();
   mem.aMethod( 10 );
   System.out.println( mem.i + mem.j );  // is this a bug?
  }}
```

new Revisited

- Member class objects can only be created if they have access to an enclosing class object
- This happens by default if the member class object is created by an instance method belonging to its enclosing class
- Otherwise it is possible to specify an enclosing class object using the **`new`** operator as follows:

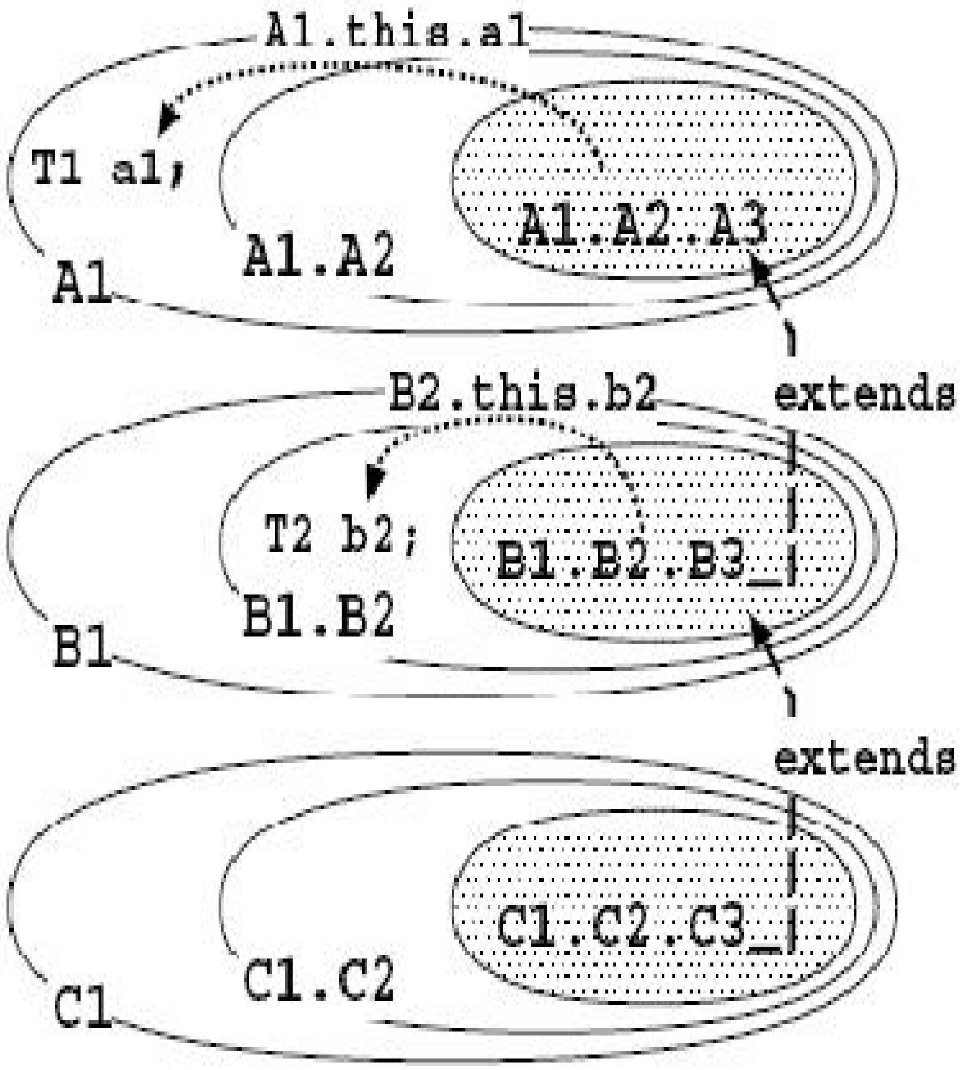
```
MemberClass b = anEnclosingClass.new MemberClass();
```

new Revisited (example)

```
public class EnclosingClass {
  ....code...
  public class MemberClass {
     ...code
    public void aMethod( int i ) {
       ...code
    }
  } // end of member class
} // end of enclosing class
class test{
 public static void main(String a[]){
    EnclosingClass ec = new EnclosingClass();
    EnclosingClass.MemberClass b; // def of the variable's type
    //b = new EnclosingClass.MemberClass(); // WILL NOT WORK
    b = ec.new  MemberClass(); // WORKS (the correcty way)
    b.aMethod(7);
 }}
```

Subclassing and Inner Classes

```
class A1 ... {
  T1 a1; ...
  class A2 ... { ...
    class A3 ... { ...
}}}
class B1 ... { ...
  class B2 ... { ...
    T2 b2; ...
    class B3 extends A1.A2.A3 {
    ...
}}}
class C1 ... { ...
  class C2 ... { ...
    class C3 extends B1.B2.B3 {
    ...
}}}
```



Another example

- To see an inner class in use, let's first consider an array. In the following example, we will create an array, fill it with integer values and then output only even values of the array in ascending order. The DataStructure class below consists of:
  - The DataStructure outer class, which includes methods to add an integer onto the array and print out even values of the array.
  - The InnerEvenIterator inner class, which is similar to a standard Java iterator. Iterators are used to step through a data structure and typically have methods to test for the last element, retrieve the current element, and move to the next element.
  - A main method that instantiates a DataStructure object (ds) and uses it to fill the arrayOfInts array with integer values (0, 1, 2, 3, etc.), then calls a printEven method to print out the even values of arrayOfInts.

```
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 even values of the array
        InnerEvenIterator iterator = this.new InnerEvenIterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.getNext() + " ");
        }
    }
//inner class implements the Iterator pattern
    private class InnerEvenIterator {
        //start stepping through the array from the beginning
        private int next = 0;
        public boolean hasNext() {
            //check if a current element is the last in the array
            return (next <= SIZE);
        }
        public int getNext() {
            //record an even element of the array
            int retValue = arrayOfInts[next];
            //get the next even element
            next += 2;
            return retValue;
        }
    } // end of inner class
    public static void main(String s[]) {
        //fill the array with integer values and print out only even numbers
        DataStructure ds = new DataStructure();
        ds.printEven();
    }
} // end of DataStructure  class
```

The output is: 0 2 4 6 8 10 12 14

- Note that the InnerEvenIterator class refers directly to the arrayOfInts instance variable of the DataStructure object.

C/ Local Classes

- A local class is a class declared within the scope of a compound statement, like a local variable
- A local class is a member class, but cannot include static variables, methods or classes. Additionally they cannot be declared **`public`**, **`protected`**, **`private`** or **`static`**
- A local class has the ability to access **`final`** variables and parameters in the enclosing scope

Local Class Example

```
public class EnclosingClass {
  String name = "Local class example";

  public void aMethod( final int h, int w ) {
    int j = 20; final int k = 30;
    class LocalClass {
      public void aMethod() {
        System.out.println( h );
        // System.out.println( w ); ERROR w is not final
        // System.out.println( j ); ERROR j is not final
       System.out.println( k );
       // System.out.println( i ); ERROR i is not declared yet
       System.out.println( name); // normal member access
    }}
    LocalClass l = new LocalClass(); l.aMethod();
    final int i = 10; } // method end
    public static void main() {
      EnclosingClass c = new EnclosingClass();
      c.aMethod( 10, 50 ); }}
```

D/ Anonymous Classes

- An anonymous class is a local class that does not have a name
- An anonymous class allows an object to be created using an expression that combines object creation with the declaration of the class
- This avoids naming a class, at the cost of being able to create only one instance of that anonymous class
- This is handy in the AWT

Anonymous Class Syntax

- An anonymous class is defined as part of a new expression and must be a subclass or implement an interface

```
new className( argumentList ) { classBody }
new interfaceName() { classBody }
```

the new class will extend/implement that class/interface

- The class body can define methods but cannot define any constructors

- The restrictions imposed on local classes also apply

Using Anonymous Classes

```
class Person {
   String name;
   public String toString() { return "My name is "+ name;}
   public Person(String nm) {name=nm;}
   public Person() {this("not specified");}
}
class PRINTER{
   void print(Object o){System.out.println(o.toString());}
}
public class AnonSimple {
   public static void main(String arg[]){
       PRINTER pr = new PRINTER();
       Person person = new Person();
       pr.print(person);
       pr.print(new Person("Yannis"));
       pr.print(new Person("Nikos"){
              public String toString() {return super.toString()+
                                                " from Crete";}
                                      }
       ); // end of method call
   } // end of main
} // end of class
```

output:

```
My name is not specified
My name is Yannis
My name is Nikos from Crete
```

Using Anonymous Classes

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MainProg {
  JFrame win;

  public MainProg( String title ) {
    win = new JFrame( title );

    win.addWindowListener(
      new WindowAdapter() {
        public void windowClosing( WindowEvent e ) {
          System.exit( 0 );
      }});
  }

  public static void main( String args[] ) {
      MainProg x = new MainProg( "Simple Example" );
    }}
```

Inner Classes in Data Structures (Linked List)

```
class LinkedList {
  Element head;
      class Element {
            Object datum;
            Element next;
            public void Extract () {
                  if (head == this)
                        head = next;
                  else {
                        Element prevPtr = head;
                        while (prevPtr.next != this)
                              prevPtr = prevPtr.next;
                        prevPtr.next = next;
                  }
            }
      }
}
```

Inner Classes in Data Structures (Map)

```
public interface Map {
  int size();
  boolean isEmpty;
  boolean containsKey(Object
  key);
  boolean containsValue(Object
                  value);
  Object get(Object key);
  Object put(Object key,
  Object value);
  Object remove(Object key);
  void putAll(Map t);
  void clear();
  public Set keySet();
  public Collection values();
  public Set entrySet();
```

```
public interface Entry {
      Object getKey();
      Object getValue();
      Object setValue(Object
            value);
      boolean equals(Object o);
      int hashCode();
      }
      boolean equals(Object o);
      int hashCode();
}
```

Inner Classes in Data Structures (Map)

```
public void printMap(Map map, PrintWriter pw) {

  Iterator entries = map.entrySet().iterator();
  while (entries.hasNext()) {
      Map.Entry entry = (Map.Entry) entries.next();
      Object key = entry.getKey();
      Object value = entry.getValue();
      pw.println(key + " -> " + value);
  }
}
```

Inner Classes in AWT programming

- Counter and event handler (`ActionListener`)

```
class Counter {
  int x=0;
  void regButton(Button b) {
    b.addActionListener(new Listener());}
  class Listener implements ActionListener {
    public void actionPerformed(ActionEvent e)
     { x++; }}}
```

- ◆Method **`regButton`** associates a listener with a given button
- ◆Every time the button is pressed, **`x`** will be incremented through the invocation of **`actionPerformed`** of the listener object

Why Inner Class?

- Each inner class can independently inherit from other classes
  - outer class inherits from one class
  - inner class inherits from another class
- The only way to get “multiple implementation inheritance”
  - Reuse code from more than one superclass
- Makes design more modular
- Reduces complexity
  - By encapsulation/information hiding

The Object-Oriented advantage

- With Inner classes you can turn things into Objects
- For example in `Member` class:
  - The Inner class to Search a Tree removes the logic of the algorithm to from an `Enclosing` class method
  - You don’t need intimate knowledge of the tree’s data structure to accomplish a search
  - From an Object-Oriented point of view, the tree is a tree and not a search algorithm

The Call Back Advantage

```
public class SomeGUI extends
JFrame implements ActionListener
{
   protected JButton button1;
   protected JButton button2;
   ...
   protected JButton buttonN;


   public void
actionPerformed(ActionEvent e)
   {
      if(e.getSource()==button1)
      {
         // do something
         }
      else
if(e.getSource()==button2)
...
```

```
public class SomeGUI extends JFrame
{
 ... button member declarations...
 protected void buildGUI()
 {
  button1 = new JButton();
  button2 = new JButton();
  ...

  button1.addActionListener(
   new java.awt.event.
              ActionListener()
    {
     public void actionPerformed
     (java.awt.event.ActionEvent e)
        {
         // do something
}});
```

Disadvantages

- Difficult to understand for inexperienced Java Developers
- Increases the total number of classes in your code
- The Developer tools does not support many things about the Inner Classes (e.g. Browsing)
- Many security issues about this subject
  - Some people insist not to use Inner classes because they can be used from anyone to alter or extend the functionality of the enclosing class or some other package class