Inner classes

Reading for these lectures:
Weiss, Section 15.1 (Iterators and nested classes), Section 15.2 (Iterators and inner classes).

A **nested class** is a **static** class that is defined inside another class. A nested class gets its own file drawer. A nested class can reference only static (and not non-static) variables of the class in which it is nested.

```
public class X {
    public static final int Y= 2;
    private int x;
    private static class SC {
        private int p;
        private static int q;
        public void m()
            {  p=  Y;  q= p; }
    }
}
```

In method m, can reference, p, q, Y, but not x.

A class that is not defined inside another can't be static

Reasons for using nested class. Get it out of the way (e.g. a once-used functor), perhaps make it private so others can't refer to it directly.

1

Inner classes

An **inner class** is a non-static class that is defined inside another class. We investigate such classes here.

Start by defining a class that allows arrays of any size, like util.Vector (but simpler, to show the idea).

```
// An array that changes to fit the required size
public class DynamicArray {

  // Constructor: an array with 10 elements allocated
  public DynamicArray() {…}

  // = the number of elements in use
  public int length() {..}

  // = the element at index i (given 0 <= i < length())
  public Object get(int i) {...}

  // set the element at position i to v (0 <= i)
  public void set(int i, Object v) {…}
}
```

2

Class DynamicArray

```
// An array that changes to fit the required size
public class DynamicArray{
    // b[0..n-1] are the elements. If n!=0, b[n-1] is the
    // element with highest index into which a value
    // was stored.
    private Object[] b;
    private int n;

    // Constructor: an array with 10 elements allocated
    public DynamicArray() {
        b= new Object[10];
        n= 0;
    }

    // = the number of elements in use
    public int length() {
        return n;
    }

    // = the element at index i.
    // Precondition: 0 <= i < length()
    public Object get(int i) {
        return b[i];
    }
```

3

Class DynamicArray (continued)

```
    // set the element at position i to v (0 <= i)
    public void set(int i, Object v) {
        if (i >= b.length) {
            // Create a new array newb with at least i+1
            // elements. For efficiency, at least double the
            // array size
                Object[] newb=
                    new Object[Math.max(i+1, 2*b.length)];

            // Copy b[0..length()] into newb
                for (int j= 0; j != length(); j++)
                    {  newb[j]= b[j];  }

            b= newb;
        }

        // {i < b.length, so v can be stored in b[i]}
        b[i]= v;
        n= Math.max(n, i+1);
    }

}
```

4

An iterator over a DynamicArray

```
import java.util.*;

public class DAIterator implements Iterator {
   private DynamicArray b; // The array to process
   private int k= 0;      // Next element to process.
                          // 0 <= k <= b.length()

   // Constructor: an iterator over b
   public DAIterator(DynamicArray b)   {  this.b= b;  }

   // = "there is another item to process"
   public boolean hasNext() { return k < b.length(); }

   // = the next item to process. Call only once per item
   //  Throw an exception if no more items exist
   public Object next() {
      if (k == b.length())
         throw new IllegalStateException(" ... ");
      k= k+1;
      return b.get(k-1);
   }

   // Not supported; does nothing
   public void remove()  {}
}
```

5

Problem with this iterator over DynamicArray

• Class DynamicArray and its iterator are separate, in two distinct files. Perhaps double what we need, and this makes a difference when there are hundreds of classes to maintain.

• Users don't have to be able to see the iterator, they just have obtain a new instance when it is needed.

Making the iterator an **inner class** is a better solution. An inner class is a non-static class that is defined inside another class.

6

DynamicArray with an inner class

```
import java.util.*;
public class DynamicArray{
   private Object[] b;          private int n;
   public DynamicArray() { … }
   public int length() { … }
   public Object get(int i) { … }
   public void set(int i, Object v) { … }

   // = an iterator over this DynamicArray
   public Iterator iterator() {return new DAIterator();}

   private class DAIterator implements Iterator {
      private int k= 0;

      public boolean hasNext() { return k < n; }

      public Object next() {
         if (k == n)  throw new ...(" .. ");
         k= k+1;   return b[k-1];
      }

      public void remove()  {}
   }
}
```

inner class

DynamicArray with an inner class

Important points about the previous slide.

- **1.** Class DAIterator is a private component of class DynamicArray --private so that outsiders can't see it.
- 2. Class DAIterator is in each instance of DynamicArray.
- **3.** An instance of DAIterator has access to the fields of the instance of DynamicArray in which it was created -- see next slide. Because of this
  - Array b can be referenced directly
  - Variable n can be referenced directly
- **4.** Field b of previous DAIterator is not needed.
- **5.** Public method iterator to create an instance of DAIterator.

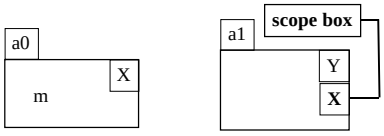
- **6**. Inner classes cannot have static components.

The memory model and nested classes (remember, a nested class is static)

An instance of a nested class has a scope box, which contains the name of class in which the nested class is defined. **Reason:** So methods in Y can access static variables of X.

Suppose a0.m() is called (see below). Instance a1 is created, and its scope box is filled with name X.

```
public class X {
   public static class Y {
   ...
   }
   public Y m() {return new Y();}
}
```



9

The memory model and inner classes (remember, an inner class is non-static)

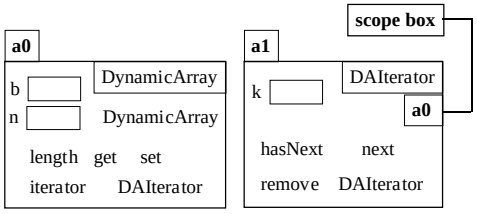
An instance of an inner class X (say) has a scope box. It contains the name Y (say) of the instance in which the inner class appears. **Reason:** So methods of X can reference fields of the instance of Y. Suppose

```
a0.iterator()
```

is called, where iterator is

```
public Iterator iterator()
   {  return new DAIterator();  }
```

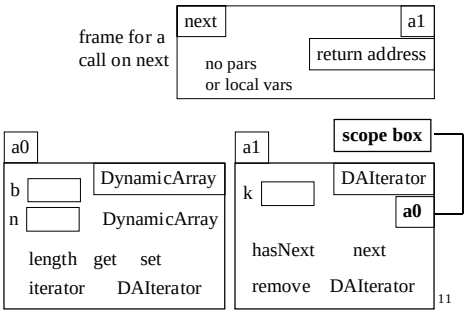
Instance a1 is created, and its scope box is filled with name a0 of the instance in which the iterator occurs.



10

Memory model: referencing a non-static name

When looking for a non-static name:
Look first in the frame.
If not there, look (in bottom-up fashion) in the object x (say) given by the scope box of the frame.
If not there, look (in bottom-up fashion) in the object y (say) given by x's scope box.
If not there, look (in bottom-up fashion) in the object given by y's scope box.
etc.

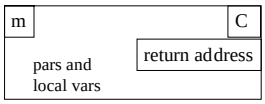


11

Memory model: referencing a static name

When looking for a static name in a frame for a call on a static method --the frame's scope box is the name of a class C: Use the algorithm in the box below.

To find a static name given a class C:
Look in class C's file drawer.
If not there, look in drawer for C's superclass C1 (say).
If not there, look in drawer for C1's superclass C2 (say).
… (continue in this fashion) ...



frame for a call on static method m

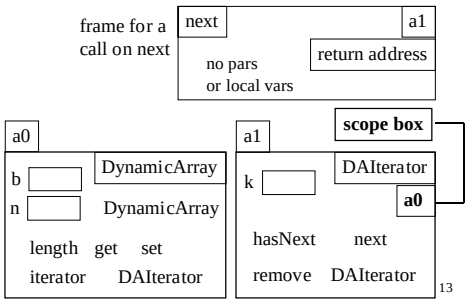
12

Memory model: referencing a static name

When looking for a static name in a frame for a call on a non-static method --the frame's scope box is the name of an object a1 (say).
An inner class cannot contain static declarations. Only inner-class objects have scope boxes. So execute this:

```
Object t= a1;
while (object t has a scope box)
   t= the name in t's scope box;
```

Now use the algorithm in the previous slide for finding a static name, using the class of object t

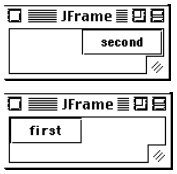


13

An example: responding to a button press in a closeable window

Reading: Weiss, sect. B.3.4. Event handling: adaptors ...

Develop a program that brings up this window on the monitor. Only one button is enabled. When the enabled button is pressed, it becomes disabled and the other becomes enabled.



Use this to show a use of inner classes and anonymous classes.

A window on the monitor corresponds to an instance of class JFrame. Get a window using

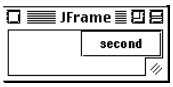
```
JFrame jf= new JFrame("title of frame");
jf.pack();   // Call after all components have been
             // added to window.
jf.show();   // Make window visible on monitor
```

Window jf has no button or other components. Clicking its close box hides the window but doesn't terminate the program.

14

An example: making window closeable

To have the program do something when close button is pressed, need to **register** a "window listener", by



(1) implementing class WindowListener

(2) providing the seven methods of that class, each of which deals with one of the boxes in the title bar of the window or with the window as a whole.

(3) Executing the following statement of method JFrame,

addWindowListener(this);

which registers the instance in which it appears as being a window listener, and

(4) putting the following statement in the method that "listens" to a press of the close box:

System.exit(0);

See the next slide.

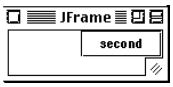
15

An example: making window closeable


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

public class CloseableFrame extends JFrame
            implements WindowListener {
   // Constructor: a Frame with good closebox, title t
   public CloseableFrame(String t) {
      super(t);
      addWindowListener(this);
   }

   // Terminate program. Called when closebox pressed
   public void windowClosing(WindowEvent e)
      { System.exit(0); }

   // Each of the following methods deals with one
   // of the window boxes or with some action on the
   // window. They don't do anything
   public void windowClosed(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowActivated(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
   public void windowOpened(WindowEvent e) {}
}
```

16

An example: making window closeable



The last slide was messy. To help out, Java provides an abstract class, like CloseableFrame, that contains empty methods for all seven methods. The class is called **WindowAdapter**.

So we would like to do the following, but it's illegal!!

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

public class CloseableFrame
        extends JFrame, WindowAdapter {

   // Constructor: a Frame with good closebox
   public CloseableFrame()
      { addWindowListener(this); }

   // Terminate program. Called when closebox pressed
   public void windowClosing(WindowEvent e)
      { System.exit(0); }
}
```

Can't extend two classes

17

An example: making window closeable

Solution to problem on previous slide:

Provide an inner class, and let the inner class extend WindowAdapter.

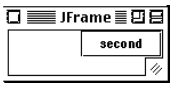
```
import javax.swing.*;
import java.awt.event.*;

public class CloseableFrame extends JFrame {

   // Constructor: a Frame with good closebox
   public CloseableFrame()
      { addWindowListener(new ExitOnClose()); }

   private class ExitOnClose extends WindowAdapter {
      // Terminate program when closebox pressed
      public void windowClosing(WindowEvent e)
         { System.exit(0); }
   }
}
```

creation of instance of inner class

18

An example: making window closeable

We can make the inner class anonymous:




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

public class CloseableFrame extends JFrame {

   // Constructor: a Frame with good closebox
   public CloseableFrame() {
      addWindowListener( new
         WindowAdapter() {
           // Terminate program when closebox pressed
           public void windowClosing(WindowEvent e)
            { System.exit(0); }
         }
      );
   }
}
```

anonymous class

An example: making window closeable

After all this, we tell you that class JFrame provides a simpler solution. Simply call JFrame's method setDefaultCloseOperation.

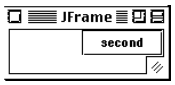

```
JFrame jf= new JFrame();
jf.setDefaultCloseOperation(
      JFrame.EXIT_ON_CLOSE);
jf.pack();
jf.show();
```

But constant EXIT_ON_CLOSE is in class JFrame only since Java 2 version 1.3, not in version 1.2.

Also, you can't use this method when using the older class Frame

So, what was said on previous slides is still useful.

Responding to a button press



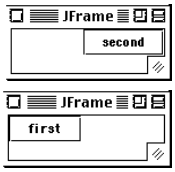
Develop a program that brings up this window on the monitor. Only one button is enabled. When the enabled button is pressed, it becomes disabled and the other becomes enabled.

Listening to a button requires implementing this interface --we need method ActionPerformed. A button press is one kind of "ActionEvent".

```
package java.awt.event;
import java.util.EventListener;

/**  Implement this interface to have a class respond to
     ActionEvents for a Component. */
public interface ActionListener extends EventListener {

   /**  Called when e happens in a Component with
        which this ActionListener is registered.
        Process a button press */
   public void actionPerformed(ActionEvent e) {
      boolean b= (be.isEnabled());
      be.setEnabled(!b);  bw.setEnabled(b);
   }
}
```

21

An example: responding to a button press



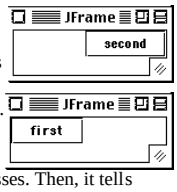
An instance of class JFrame is a window on your monitor. An instance of JButton is a Component that can be placed in a JFrame.

Method main creates a new JFrame. We have to show what a constructor does and what actionPerformed does.

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

public class Applic extends JFrame
                    implements ActionListener{
   private JButton bw= new JButton("first");
   private JButton be=  new JButton("second");

   public static void main(String pars[])
   {  Applic jF= new Applic("JFrame");  }

   // Constructor: a frame: two buttons, title t
   public Applic(String t) { ... }

   public void actionPerformed(ActionEvent e) { ... }
};
```

22

An example: responding to a button press



The constructor first calls the constructor of the superclass, giving it the title for the window. It then adds the two buttons to the window -- getContentPane is a JFrame method. It enables one button and disables the other. And it registers this instance as a "listener" for button presses. Then, it tells the JFrame to place all components. And its makes the window visible. Isn't that easy?

```
// Constructor: an Applic with two buttons and title t
public Applic(String t) {
    super(t);
    getContentPane().add(bw, BorderLayout.WEST);
    getContentPane().add(be, BorderLayout.EAST);

    bw.setEnabled(false);
    be.setEnabled(true);

    // Set the actionlistener for the buttons
        bw.addActionListener(this);
        be.addActionListener(this);

    pack();
    setVisible(true);
}
```

23

An example: responding to a button press

We can hide things by using an anonymous class.

```
public class Applic extends JFrame {
    private JButton bw= new JButton("first");
    private JButton be=  new JButton("second");

    // Constructor: an Applic with two buttons and title t
    public Applic(String t) {super(t);
        getContentPane().add(bw, BorderLayout.WEST);
        getContentPane().add(be, BorderLayout.EAST);

        bw.setEnabled(false);
        be.setEnabled(true);

        // Set the actionlistener for the buttons
            bw.addActionListener(al);
            be.addActionListener(al);

        pack();  setVisible(true);
    }

    private ActionListener al= new ActionListener() {
        // Process a button press
        public void actionPerformed(ActionEvent e) {
            boolean b= (be.isEnabled());
            be.setEnabled(!b);  bw.setEnabled(b);
        }
    };
}
```

24