

CSE 8B—Intro to CS: Java

Winter, 2006

February 23 (Day 14)

Menus

Swing Event Handling

Inner classes

Instructor: Neil Rhodes

Menus

JMenuBar

- container for menus
- associated with a JFrame via JFrame.setMenuBar

JMenu

- container for menu items
- has a title

JMenuItem

- Has a title
- Can have an icon, checkmark, shortcut

Menu Sample

```
public class MenuTest extends JFrame {
    public static void main(String[] args) {
        MenuTest window = new MenuTest();
        window.setVisible(true);
    }

    MenuTest() {
        super("MenuTest");
        setSize(300, 100);

        JButton button1 = new JButton("Button 1");
        add(button1);

        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        JMenu menu1 = new JMenu("File");
        menu1.add(new JMenuItem("New"));
        menu1.add(new JMenuItem("Open"));
        menu1.add(new JMenuItem("Close"));
        menuBar.add(menu1);

        JMenu menu2 = new JMenu("Edit");
        menu2.add(new JMenuItem("Cut"));
        menuBar.add(menu2);
    }
}
```

3

Using Inner Classes

We've seen using

- The subclass of JFrame as a listener
- Another class in a different file as a listener

Sometimes, want a class dedicated as a listener, but not in a different file

- Different file means listener doesn't have access to private variables in the subclass of JFrame
- Different file means trudging across files to try to figure out GUI logic

Inner classes

- Defined in another (outer) class
- Have access to instance variables of outer class

4

Example of Inner class

```
public class MenuTest2 extends JFrame {
    private class CloseOnActionListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            dispose(); // executes dispose on JFrame, since no dispose
                       // is defined for CloseOnAction
        }
    }
    public static void main ...
    MenuTest2() {
        ...

        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        menu1.add(new JMenuItem("Open"));
        JMenuItem close = new JMenuItem("Close");
        close.addActionListener(new CloseOnActionListener());
        menu1.add(close);
        menuBar.add(menu1);
    }
}
```

5

Inner class can be local to a method

Class still has access to all instance variables of the outer class

- Local variables are limited: only access to local variables is final ones.

Advantage:

- Moving code of class even closer to where it is needed
- Always an advantage to make things as local as possible

Disadvantage:

- Can't reuse class outside of the method it's in

6

Example of local inner class

```
public class MenuTest4 extends JFrame {
    public static void main(String[] args) {
        MenuTest4 window = new MenuTest4();
        window.setVisible(true);
    }
    MenuTest4() {
        super("MenuTest4");
        final Button button = new Button("Testing");
        add(button);
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        JMenu menu1 = new JMenu("File");
        JMenuItem close = new JMenuItem("Disable Button");
        class DisableButton implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                button.setEnabled(false);
            }
        }
        close.addActionListener(new DisableButton());
        menu1.add(close);
        menuBar.add(menu1);
    }
}
```

7

Getting rid of unnecessary names

We can do either:

- `Person p = new Person("John", "Smith");`
`addPerson(p);`

or:

- `addPerson(new Person("John", "Smith"));`

In one case, we have a named variable containing the person. In the other case, we have an *anonymous* person.

Java allows the same idea with classes: *anonymous classes*

- Define a new class with no name
- Can only use it to instantiate an object of that type
- Can't instantiate more than one object or reuse the class

8

Anonymous classes

Syntax

- `new SuperClass(args to SuperClass constructr){`
 non-static methods
 non-static instance variables
}

Usage:

- Use it wherever you can provide an object:
 - `addPerson(new Person("Joe", "Jones") {`
 public static getAge() { return super.getAge() /
 2; }
});

Limitations:

- No constructors
- No static methods/variables
- No extends or implements

9

Very useful with listeners

```
public class MenuTest5 extends JFrame {
    public static void main(String[] args) {
        MenuTest5 window = new MenuTest5();
        window.setVisible(true);
    }
    MenuTest5() {
        super("MenuTest5");
        final Button button = new Button("Testing");
        add(button);
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        JMenu menu1 = new JMenu("File");
        JMenuItem close = new JMenuItem("Disable Button");
        close.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                button.setEnabled(false);
            }
        });
        menu1.add(close);
        menuBar.add(menu1);
    }
}
```

10

Anonymous interfaces

If the specified SuperClass is actually an interface, the class created is a subclass of Object that implements the given interface

```
new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        button.setEnabled(false);  
    }  
}
```

11

Adapters

Consider WindowListener

- Has 7 methods:
 - `windowOpened`
 - `windowClosing`
 - `windowClosed`
 - `windowIconified`
 - `windowDeiconified`
 - `windowActivated`
 - `windowDeactivated`
- What if you want to listen just for the window about to close?
 - Requires 6 empty methods and one real one:
 - ```
public class MyListener implements WindowListener {
 // six empty methods
 public void windowClosing(WindowEvent e) {
 dispose(); //close the JFrame
 if (--numWindows == 0)
 System.exit(0);
 }
}
```

12

## WindowAdapter

An adapter is a class that implements a do-nothing listener interface.

Swing provides a `WindowAdapter`:

```
▪ public class WindowAdapter implements WindowListener {
 public void windowOpened(WindowEvent e) {}
 public void windowClosing(WindowEvent e) {}
 public void windowClosed(WindowEvent e) {}
 public void windowIconified(WindowEvent e) {}
 public void windowActivated(WindowEvent e) {}
 public void windowDeactivated(WindowEvent e) {}
}
```

Your class now doesn't need 6 empty methods:

```
public class MyListener extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
 dispose(); //close the JFrame
 if (--numWindows == 0)
 System.exit(0);
 }
}
```

13

## Coding Style

Make small, self-contained methods

- 25-line methods are almost always too long

Use naming to self-document

- Fewer comments
- Better structure and naming

Compare

```
public void doIt() {
 // prep the input and output
 Scanner ...
 ...

 // read the input
 while (scanner.hasNextLine())
 ...

 // sort the array
 Arrays.sort(...)

 // write the output
 for (...)
 ...
}
```

```
public void doIt() {
 prepInputAndOutput();
 readInput();
 sortArray();
 writeOutput();
}

private void prepInputAndOutput()
 {...}
private void readInput() {...}
private void sortArray() {...}
private void writeOutput() {...}
```

14