User Interface Programming

OOP/Java Primer
Step 4 – Some Final Checkups
Classes, Inner and Anonymous

- Sometimes we encounter classes within other classes.

- Why should we define inner and anonymous classes at all?

- They are needed!

(http://blogs.sun.com/darcy/entry/nested_inner_member_and_top)
Privileges

- Non-static inner classes have access to other instances of the enclosing class
  - even if they are declared private.

- Static nested classes do **not** have access to other instances of the enclosing class

- An Instance of InnerClass Exists Within an Instance of OuterClass
Reasons

- **Logical grouping of classes**
  - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.
  - Nesting such "helper classes" makes their package more streamlined.

- **More readable, maintainable code**
  - Nesting small classes within top-level classes places the code closer to where it is used.
Reasons

- **Increased encapsulation**
  - Consider two top-level classes, A and B, where B needs access to instances of A that would otherwise be declared private.
  - By hiding class B within class A, A's instances can be declared private and B can access them.
  - In addition, B itself can be hidden from the outside world.
Inner (nested) classes

- Java does not support multiple inheritance

- If your class inherits another class, e.g. JFrame it can’t inherit an Adapter class

- Solution: use an inner class that inherits the Adapter class
public class MyFrame extends JFrame {  // is a-relation
...
anObject.addMouseListener(new MyAdapter());
...

class MyAdapter extends MouseAdapter {  // extends adapter
    public void mouseClicked(MouseEvent e) {
        ... // Some implementation code
    }
}
}
Local Inner Classes

- Are defined within the body of a method

- These classes are used in the same way as anonymous classes
  - May have constructors
public class LocalInnerClass {
    private static Test monitor = new Test();
    private int count = 0;

    Counter getCounter(final String name) {
        // A local inner class:
        class LocalCounter implements Counter {
            public LocalCounter() { // Constructor !
                System.out.println("LocalCounter()");
            }
            public int next() { // Method
                return count++;
            }
        }
        return new LocalCounter(); // Returns an instance
    }

    // of LocalCounter
Anonymous class

- An inner class that is defined where it is used
  - + Makes the code easier to read
  - + The class is in its proper place
  - - Longer code structures
  - - No Constructors
Example

```java
public class LocalInnerClass {
    private static Test monitor = new Test();
    private int count = 0;

    Counter getCounter2(final String name) {
        return new Counter() {
            //instance initializer instead of constructor
            { System.out.println("Counter()"); }
            public int next() {
                return count++;
            }
        };
    }
}
```
Problems

- More classes - affects efficiency
- Can become "messy" if overused
- Needs to be well documented
More MVC

- Recap and correction… 😞
First

- Perfect MVC is not possible
- MVC is an aim – not an absolute measure
- It has a purpose
Purpose

- Increase encapsulation

- Separation of model and view
  - Separation of representation and presentation

- But control?
  - As far as possible
Control needs to:

- be provided for the Listener exchange in view

```java
void addMultiplyListener(ActionListener mal) {
    m_multiplyBtn.addActionListener(mal);
}

void addClearListener(ActionListener cal) {
    m_clearBtn.addActionListener(cal);
}
```

- and then add the listeners in control:

```java
//... Add listeners to the view.
view.addMultiplyListener(new MultiplyListener());
view.addClearListener(new ClearListener());
```
Control needs to know:

- about the available actions in the model

```java
public void multiplyBy(String operand) {
    m_total = m_total.multiply(new BigInteger(operand));
}
```

- And call them

```java
try {
    userInput = m_view.getUserInput();
    m_model.multiplyBy(userInput);
    m_view.setTotal(m_model.getValue());
}
```

```java
} catch (NumberFormatException nfex) {
    m_view.showError("Bad input: "+ userInput + ":");
}
```
Mikael’s example

- $M - (C - C) - V$

- Separation of $M - C-C - V$

- $C$ is divided into two communicating parts

- One $C$ is directed towards $M$ and the other towards $V$
Mikael’s example

- Separation of V and M shows possibility to have several views for one model.
- Also the different views can trigger the control and promote a change in the model.
Where is Control?

- User interface?
- Keyboard?
- Joystick?
- Virtual Joystick?
Actual – perceived

- Where lies the control?

- JButton – ActionListener => surfacial level
Actions

- A manifestation of the control-aspect

- Not in the interface
  - But sometimes placed in the UI-package
  - Gets signals from the UI

- Not in the model
  - But has to know what he model can do
Actions

- More hardcoded
- Less handling
- Less control
- Solid solution
View… Windows

- Window sizing and placement is sometimes…

Surprising

- … to say the least…
JFrame Sizing

- Suppose that we want a Window to show a picture with size 100 * 100 pixels

```java
JFrame frame = // ...
frame.setSize(100,100); // make frame 100x100
```

- Wrong!

- A JFrame has decorations!
  - Decorations may also use different space on different platforms
JFrame Sizing

- Image Size → painted on a JPanel

- Set Panel Size,

- Put Panel in JFrame

- Use `pack()`
  - `pack()` makes frame as small as the components will allow
Fixed Size Window

- To get a window that has a fixed size
  - Cannot be changed by the user (!)

```java
frame.setResizable(false);
```

- Only do this when you design a window that MUST not be changed
Minimum Size?

- Is it possible to set a minimum size for a Window?

- Unfortunately not (!)

- There is a fix for doing this…
ComponentListeners

- Not so elegant, but it works

```java
final JFrame frame = // ...

    // Anonymous class
frame.addComponentListener(new java.awt.event.ComponentAdapter() {
    public void componentResized(ComponentEvent event) {
        frame.setSize(
            Math.max(100, frame.getWidth()),
            Math.max(100, frame.getHeight()));
    }
});
```
setPreferredSize

- setPreferredSize() works together with pack()

- The general advice is:
  - Don’t change layout after initialization
  - Initialization is made through pack()
  - setVisible(true) after pack()
Swing problems in general:

- Sometimes Swing problems are difficult to debug
- Often because of an immense constructor!
- All initializations don’t need to be in one method…

Refactor!!!!
How to start the program?

- Don’t just create the JFrame class with a main method that calls the constructor!
  - The reasons for this will become clear later!
    - Short reason: The JFrame is NOT your application, it is a window!
  - Make a separate class that serves as the controlling spider in the net.
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
            new myProgram.makeUI();
        }
    });
}