Nested Classes

Summary

- A (kind of awkward) solution for assignment 7's extra challenge.
- A better solution using a normal (not nested) class that implements the Comparable interface.
- Another version of the better solution, with a *static nested class*.
- A simple GUI program with a normal (not nested) class extending JPanel.
- A second version of the GUI program, with an *inner class* (a non-static nested class) extending JPanel.
- An interactive version of the GUI program, with an *inner class* extending MouseAdapter.
- A fourth version, with an *anonymous inner class* extending MouseAdapter.

Assignment 7 Extra Challenge

The extra challenge for assignment 7 (the substitution cipher assignment) was to add to your program the capability of decoding messages encoded with either of the first two encoding schemes. The idea was to use "frequency analysis": to count how many times each character appears in the encoded text, sort characters from most to least common, and match them up with a ranking based on normal (not encoded text) to figure out which plaintext character each encoded text character represents.

The tricky part is that you needed to sort the encoded text character frequencies in a way that preserves the association between the character and the number of times it occurs. Here's one way to do it: make an array of Strings, where each element of the array consists of a number and a character, e.g.,

`["08 A", "19 B", "13 C", "02 D"...`

You could then use the `Arrays.sort` method to sort this array, and because of the fact that the numbers come first (and smaller numbers have leading zeros) you end up with the characters in the right order. Here's a full `decode` method making use of this approach:

```
public static String decode(String cipherText, char[] ranking) {

    // Make String array with counts and character for each
    // count value (e.g., ["14 A", "04 B", "03 C", ...]).

    int[] counts = countCharacters(cipherText);
    String[] countsWithCharacters = new String[26];

    int maxCount = 0;

    for (int i : counts) {
        if (i > maxCount) {
            maxCount = i;
        }
    }

    int maxDigits = (int) Math.log10(maxCount) + 1;
    String formatString = String.format(
            "%%0%dd %%c",  // %% for literal % sign in String.
            maxDigits);

    for (int i = 0; i < 26; i++) {
        countsWithCharacters[i] = String.format(formatString,
                counts[i], (char) (i + 'A'));
    }

    // Sort counts, keeping associated character with count,
```

```
    // since it's part of the same String.  Then strip counts
    // to get an array of chars representing the ranking of
    // characters for cipherText input.  Ranking order should be
    // opposite sort order (highest to lowest rather than lowest
    // to highest).

    Arrays.sort(countsWithCharacters);
    char[] cipherTextRanking = new char[26];

    for (int i = 0; i < 26; i++) {
        char c = countsWithCharacters[25 - i].charAt(
                maxDigits + 1);
        cipherTextRanking[i] = c;
    }

    // Make substitution alphabet based on matching up
    // cipherText character ranking and supplied ranking
    // (second parameter above).

    char[] alphabet = new char[26];

    for (int i = 0; i < 26; i++) {
        alphabet[cipherTextRanking[i] - 'A'] = ranking[i];
    }

    // Use substitution alphabet to decode String.

    char[] characters = cipherText.toCharArray();

    for (int i = 0; i < characters.length; i++) {
        if (characters[i] >= 'A' && characters[i] <= 'Z') {
            int j = characters[i] - 'A';
            characters[i] = alphabet[j];
        }
    }

    return String.valueOf(characters);
}
```

That's one way to do it, but there's a more elegant way to solve a problem like this in Java. You can define a class with a `compareTo` method in which you specify what it means for one instance of the class to "come before" another, and then `Arrays.sort` can be used to sort an array of instances based on your `compareTo` method. Here's a class to associate an `int` and a `char`, with a `compareTo` method written so that if you sort an array of instances they'll go from the one with the highest `int` value to the one with the lowest.

```
public class CountAndCharacter
        implements Comparable<CountAndCharacter> {
    private int count;
    private char character;

    public CountAndCharacter(int count, char character) {
        this.count = count;
        this.character = character;
    }

    @Override
```

```
    public int compareTo(CountAndCharacter that) {
        return -Integer.compare(this.count, that.count);
    }
}
```

Notice the minus sign in the `return` statement at the end. That's what makes CountAndCharacter objects sort from highest (`count`) to lowest rather than lowest to highest.

This is a short and oddly-named class, one that wouldn't really make sense except in this particular program, where it's just the thing we need. Do we really have to put it in its own file? No. We can define it inside of the class that uses it, so that it's only accessible within that class:

```
public class SubstitutionCipher {

 ...private static class CountAndCharacter
            implements Comparable<CountAndCharacter> {
        private int count;
        private char character;

        public CountAndCharacter(int count, char character) {
            this.count = count;
            this.character = character;
        }

        @Override
        public int compareTo(CountAndCharacter that) {
            return -Integer.compare(this.count, that.count);
        }
    }

    public static String decode(String cipherText, char[] ranking) {

        // Create array of CountAndCharacter objects; each element
        // associates a character with the number of times it
        // appears in cipherText.  Sort the array (from highest
        // count to lowest count, since that's the way
        // CountAndCharacter's compare method is written).

        int[] counts = countCharacters(cipherText);
        CountAndCharacter[] countsWithCharacters =
                new CountAndCharacter[26];

        for (int i = 0; i < 26; i++) {
            countsWithCharacters[i] = new CountAndCharacter(
                    counts[i], (char) (i + 'A'));
        }

        Arrays.sort(countsWithCharacters);

        // Make substitution alphabet based on matching up
        // cipherText character ranking and supplied ranking
        // (second parameter above).

        char[] alphabet = new char[26];

        for (int i = 0; i < 26; i++) {
            int j = countsWithCharacters[i].character - 'A';
```

```
            alphabet[j] = ranking[i];
        }

        // Use substitution alphabet to decode String...
```

Because CountAndCharacter is `private`, it's only accessible within the SubstitutionCipher class. Other than that it's defined and used just like a normal (not nested) class would be. (Notice that CountAndCharacter is also `static`. This means it won't be associated with any particular instance of SubstitutionCipher, just as a normal class defined in its own file would not be.)

A Simple GUI Program

Here's a class that extends JPanel, to be used in a simple GUI program.

```
public class FancyPanel extends JPanel {
    private Color color;
    private int numBoxes;
    private int width;
    private int height;

    public FancyPanel(Color color, int numBoxes,
            int width, int height) {
        this.color = color;
        this.numBoxes = numBoxes;
        this.width = width;
        this.height = height;
        setPreferredSize(new Dimension(width, height));
    }

    @Override
    public void paint(Graphics g) {
        int w = width / numBoxes;
        int h = height / numBoxes;
        Color c = color;

        for (int i = 0; i < numBoxes; i++) {
            g.setColor(c);
            g.fillRect(w / 2 * i, h / 2 * i,
                    width - w * i, height - h * i);
            c = c.brighter();
        }
    }
}
```

To understand this code, you need to know when `paint` is called. It's called any time the JVM determines that the window display needs to be refreshed. For example, if another window is moved in front of this program's window, and then moved away; or if this window is minimized and restored; etc. So whatever we put in `paint` (in our version, which overrides the default version defined in JPanel) will happen any time the window display is (re)drawn. The version of `paint` above draws a series of ever smaller and brighter rectangles centered in the window.

A class like FancyPanel would typically be nested inside a class extending JFrame, not just because FancyPanel is short and not likely to be useful in other programs, but because it can be defined in such a way that it has access to the enclosing class's fields. This eliminates the need for FancyPanel to have fields with copies of values that are likely stored in the enclosing class's fields anyway. And it shortens the code, since we don't need to copy values in the constructor. Here's the new version of FancyPanel, within an enclosing class that extends JFrame, FancyFrame.

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

@SuppressWarnings("serial")
public class FancyFrame extends JFrame {

    private final int width;
    private final int height;
    private final Color color;
    private final int numBoxes;

    public FancyFrame(String title, int width, int height,
            Color color, int numBoxes) {
        super(title);

        this.width = width;
        this.height = height;
        this.color = color;
        this.numBoxes = numBoxes;

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(new FancyPanel());
        pack();
        setVisible(true);
    }

    private class FancyPanel extends JPanel {

        public FancyPanel() {
            setPreferredSize(new Dimension(width, height));
        }

        @Override
        public void paint(Graphics g) {
            int w = width / numBoxes;
            int h = height / numBoxes;
            Color c = color;

            for (int i = 0; i < numBoxes; i++) {
                g.setColor(c);
                g.fillRect(w  / 2 * i, h / 2 * i,
                        width - w * i, height - h * i);
                c = c.brighter();
            }
        }
    }

    public static void main(String[] args) {
        new FancyFrame("Fancy!", 320, 240,
                new Color(25, 50, 75), 7);
    }
}
```

Notice that this time the nested class is *not* `static`. This means that, unlike the `static` nested class in the SubstitutionCipher example above, code within FancyPanel is associated with a specific instance of FancyFrame and has access to FancyFrame's fields. A non-`static` nested class, like FancyPanel, is also called an "inner class."

An Interactive GUI Program

Here's a new version of FancyFrame, this time with an additional inner class extending MouseAdapter. Just like FancyPanel, MousePresser is not `static`. So it has access to FancyFrame's `mousePressed` field, which is used to determine whether the `paint` method's rectangles get successively brighter or darker.

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.JFrame;
import javax.swing.JPanel;

@SuppressWarnings("serial")
public class FancyFrame extends JFrame {

    private final int width;
    private final int height;
    private final Color color;
    private final int numBoxes;

    private boolean mousePressed;

    public FancyFrame(String title, int width, int height,
            Color color, int numBoxes) {
        super(title);

        this.width = width;
        this.height = height;
        this.color = color;
        this.numBoxes = numBoxes;
        mousePressed = false;

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(new FancyPanel());
        pack();
        setVisible(true);
    }

    private class mousePresser extends MouseAdapter {

        @Override
        public void mousePressed(MouseEvent e) {
            mousePressed = true;
            repaint();
        }

        @Override
        public void mouseReleased(MouseEvent e) {
            mousePressed = false;
```

```
            repaint();
        }
    }

    private class FancyPanel extends JPanel {

        public FancyPanel() {
            addMouseListener(new mousePresser());
            setPreferredSize(new Dimension(width, height));
        }

        @Override
        public void paint(Graphics g) {
            int w = width / numBoxes;
            int h = height / numBoxes;
            Color c = color;

            for (int i = 0; i < numBoxes; i++) {
                g.setColor(c);
                g.fillRect(w  / 2 * i, h / 2 * i,
                        width - w * i, height - h * i);

                if (mousePressed) {
                    c = c.brighter();
                } else {
                    c = c.darker();
                }
            }
        }
    }

    public static void main(String[] args) {
        new FancyFrame("Fancy!", 320, 240,
                new Color(50, 80, 120), 5);
    }
}
```

Notice the calls to `repaint` in MousePresser's methods. This is the proper way to request that the window be redrawn, rather than calling the `paint` method directly.

There's another type of nested class you will often see if you look at Java GUI programming examples. (I prefer the structure above, because I think it's easier to understand, but this is an alternative you should be aware of.) In the code below, the MousePresser class definition has been moved to the place where `addMouseListener` is called. And the name "MousePresser" is gone—this is an example of an *anonymous inner class*. This pattern is typically used when the class definition is short, and it's only needed in one place.

```
public class FancyFrame extends JFrame {

 ...private class FancyPanel extends JPanel {

        public FancyPanel() {

            addMouseListener(new MouseAdapter(){

                @Override
                public void mousePressed(MouseEvent e) {
                    mousePressed = true;
```

```
            repaint();
        }

        @Override
        public void mouseReleased(MouseEvent e) {
            mousePressed = false;
            repaint();
        }
    });

    setPreferredSize(new Dimension(width, height));
}

@Override
public void paint(Graphics g) {...
```