Contents

0 What is ahead?

Java allows you to create classes inside of other classes and, even inside of methods. Creating classes in these places gives us access to the outer class's state variables. We will find that this technique is very useful for GUI programming and for creating our own *data structures*. While we are in this chapter, we learn about Java's graphics libraries and we will make our programs responsive to mouse and keyboard activity.

1 Improving Tricolor

We are going to study the `Tricolor` application and improve it via the use of inner classes.

Let us begin with the quit menu item. We have a whole external class in our project that is used in one place: on the quit menu item. Can we get rid of this complexity and place the code necessary to drive the menu item inside of `Tricolor.java`?

The answer is, "yes." What we will do is to create an *anonymous inner class*. This is a class with no name. Go into the `makeMenus` method of `Tricolor.java` and find the lines on which the quit item is created and its action listener attached.

```
JMenuItem quitItem = new JMenuItem("Quit");
fileMenu.add(quitItem);
quitItem.addActionListener(new QuitListener());
```

We will now obviate the need for the external `QuitListener` class. Change the code as follows.

```
JMenuItem quitItem = new JMenuItem("Quit");
fileMenu.add(quitItem);
quitItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
});
```

Then by add these two import statements to the beginning of the `Tricolor.java`.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

Now close the `QuitListener` class, compile, and run. You will see that the quit menu item is still working.

2 Deconstructing this Arabesque

You can see we have added some very mysterious code to `quitItem`. What does it all mean. Notice what is inside of the parentheses.

```
                            new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
}
```

You see a call to `new`, so you know an object is being created. You also know that `ActionListener` is an interface, so we can never create an instance of `ActionListener`.

So, what is happening? This is an example of an *anonymous inner class*. It is a class with no name. What you are saying here is, "make an instance of an `ActionListener` with this `actionPerformed` method." Since we never refer to it after we add it to the `quitItem` menu item object, we never do need to name it.

Anonymous inner classes provide a quick and easy way to attach actions to menu item or buttons. This is especially true if the menu item performs an isolated function, such as shutting an application down. The entire anonymous inner class is an argument you are sending to the call `quitItem.addActionListener()`!

We have now obviated the need for the `QuitListener` class, which just adds another name for a single-use object.

Take note of the way we formatted the code.

```
quitItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
});
```

Observe that the entire class declaration

```
new ActionListener()
{
   public void actionPerformed(ActionEvent e)
   {
        currentColor = Color.red;
        leftPanel.setColor(currentColor);
        leftPanel.repaint();
    }
}
```

is part of the argument in the call `quitItem.addActionListener(....);`. This explains the seemingly strange construct `\tt });`. In one shot we have implemented an object's class and instantiated it as well.

We will refer to this funny closing object,`\tt });`, as "sad Santa." If you format this way, you should see Sad Santa as the last line of an anonymous listener class. It is very important to be fanatically consistent in this matter. It helps you avoid mysterious error messages that will vex and confuse.

3 Hammertime

Just because you have a shiny new hammer the entire world *does not* become a nail. You might be tempted to do this. And if you succumb to this impulse, it is going to work. You should go ahead and try it!

```
red = new ColorMenuItem(Color.RED, "red", this);
red.addActionListener(new ActionListener()
   {
       public void actionPerformed(ActionEvent e)
       {
            currentColor = Color.red;
            currentPanel.setColor(currentColor);
            currentPanel.repaint();
        }
   });
```

You can now repeat this procedure for the green and blue menus as follows.

```
green = new ColorMenuItem(Color.GREEN, "green", this);
green.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        currentColor = Color.green;
        leftPanel.setColor(currentColor);
        leftPanel.repaint();
    }
});
blue = new ColorMenuItem(Color.BLUE, "blue", this);
blue.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        currentColor = Color.blue;
        leftPanel.setColor(currentColor);
        leftPanel.repaint();
    }
});
```

But you can see this is the sort of violation of the 11th commandment that you were earlier warned of.

4 Using Inner Classes to Improve our Design

The implementation we just showed works, but it exhibits duplicate and slack code in the implementation of the action listeners. We begin to think: *Can we attach the action listener directly to the color menu item?*. Let us return to our original design for the `ColorMenuitem` We can convert it to being an inner class to `Tricolor` and obviate the need for the `Tricolor tc` state variable. So we begin with this.

```
import javax.swing.JMenuItem;
import java.awt.Color;

public class ColorMenuItem extends JMenuItem
{
    private final Color color;
    private final Tricolor tc;
    public ColorMenuItem(Color _color, String _colorName, Tricolor _tc)
    {
        super(_colorName);
        color = _color;
        tc = _tc;
        addActionListener(new ColorMenuItemListener(tc, color));
    }
}
```

Now we trim this down to be an inner class. Both `JMenuItem` and `Color` are imported so we can lop the imports off.

Since we are going to make this into an inner class of `Tricolor`, we will have access to `Tricolor`'s state variables. As a result, we will get rid of the instance of `Tricolor` that is a state variable inside of the class. We will also get rid of the call to `ColorMenuItemListener`. This will be replaced by an inner class.

```
class ColorMenuItem extends JMenuItem
{
    private final Color color;
    public ColorMenuItem(Color _color, String _colorName)
    {
        super(_colorName);
        color = _color;
        addActionListener(); TODO:  Write action listener!
    }
}
```

Now let us pop this inside of `Tricolor`. We will need to do some adjustment before this will compile. First, change the lines in `makeMenus`

```
colorMenu.add(new ColorMenuItem(Color.RED, "red", this));
colorMenu.add(new ColorMenuItem(Color.GREEN, "green", this));
colorMenu.add(new ColorMenuItem(Color.BLUE, "blue", this));
```

to

```
colorMenu.add(new ColorMenuItem(Color.RED, "red"));
colorMenu.add(new ColorMenuItem(Color.GREEN, "green"));
colorMenu.add(new ColorMenuItem(Color.BLUE, "blue"));
```

Also add these two imports.

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
```

Our class now looks like this. You can close the `QuitListener` and `ColorMenuItem` classes. These are now obviated.

```
import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;


public class Tricolor extends JFrame implements Runnable
{
    ColorPanel leftPanel;
    ColorPanel rightPanel;
    ColorPanel middlePanel;
    ColorPanel currentPanel;

    public Tricolor()
    {
        super("Tricolor");
        leftPanel = new ColorPanel();
        middlePanel = new ColorPanel();
        rightPanel = new ColorPanel();
        currentPanel = leftPanel;
    }
    public ColorPanel getCurrentPanel()
```

```
{
    return currentPanel;
}
public void setCurrentPanel(ColorPanel c)
{
    currentPanel = c;
}
public void run()
{
    setSize(500,500);
    makeMenus();
    //install Color Panels
    Container c = getContentPane();
    c.setLayout(new GridLayout(1,3));
    c.add(leftPanel);
    c.add(middlePanel);
    c.add(rightPanel);
    setVisible(true);
}
private void makeMenus()
{
    //make and install menu bar
    JMenuBar mbar = new JMenuBar();
    setJMenuBar(mbar);
    //File menu
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
    JMenuItem quitItem = new JMenuItem("Quit");
    fileMenu.add(quitItem);
    quitItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            System.exit(0);
        }
    });
    // Color Menu
    JMenu colorMenu = new JMenu("Color");
    mbar.add(colorMenu);
    colorMenu.add(new ColorMenuItem(Color.RED, "red"));
    colorMenu.add(new ColorMenuItem(Color.GREEN, "green"));
    colorMenu.add(new ColorMenuItem(Color.BLUE, "blue"));
    JMenu positionMenu = new JMenu("Position");
    // Position Menu
    mbar.add(positionMenu);
    positionMenu.add(new PositionMenuItem(this, leftPanel, "left"));
    positionMenu.add(new PositionMenuItem(this, middlePanel, "middle"));
```

```
        positionMenu.add(new PositionMenuItem(this, rightPanel, "right"));
    }
    class ColorMenuItem extends JMenuItem
    {
        private final Color color;
        public ColorMenuItem(Color _color, String _colorName)
        {
            super(_colorName);
            color = _color;
            //addActionListener(); //TODO:  Write action listener!
        }
    }
    public static void main(String[] args)
    {
        Tricolor t = new Tricolor();
        javax.swing.SwingUtilities.invokeLater(t);
    }
}
```

Now let us get the action listener working. We will attach this as an anonymous inner class inside of `ColorMenuItem`. Begin by creating a shell for it.

```
public ColorMenuItem(Color _color, String _colorName)
{
    super(_colorName);
    color = _color;
    addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
        }
    });
}
```

Before doing anything else, compile and make sure you have all of your formatting ducks in a row. Now we write the body. What do we want to happen? The current panel should be set to this menu item's color. We should then update the graphics. You now insert this code.

```
public ColorMenuItem(Color _color, String _colorName)
{
    super(_colorName);
    color = _color;
    addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            currentPanel.setColor(color);
```

```
            repaint();
        }
    });
}
```

Now you run this shiny new code and uh oh... we appear to have a repaint error. What happened? Only the menu item is repainting itself! There are two ways to handle this. One is to tell the current panel to repaint. Change

```
repaint();
```

to

```
currentPanel.repaint();
```

Another way is to tell the whole app to repaint. You do have access to the `this` of an enclosing class. In this case just use

```
Tricolor.this.repaint();
```

We use the former solution, since it does the minimum work needed and accomplishes the goal. The color menu is now completely operational as it was before. We have relieved our code of some complexity. Since the state variables of the enclosing class are visible we can shorten our constructor and the complexity of the code inside of the `ColorMenuItem`. We make the listener a totally anonymous class, and cut complexity there too. You no longer need the two external classes that controlled the Color menu.

5 The Position Menu

Now we are ready to adjust the position menu. Here is the current state of `PositionMenuItem.java`.

```
import javax.swing.JMenuItem;
public class PositionMenuItem extends JMenuItem
{
    public final Tricolor tc;
    public final ColorPanel attachedPanel;
    public PositionMenuItem(Tricolor _tc, ColorPanel _attachedPanel,
        String pos)
    {
        super(pos);
        tc = _tc;
```

```
        attachedPanel = _attachedPanel;
        addActionListener(new PositionMenuItemListener(tc, attachedPanel));
    }
}
```

We now slim this down to be an inner class.

```
public class PositionMenuItem extends JMenuItem
{
    public final ColorPanel attachedPanel;
    public PositionMenuItem(ColorPanel _attachedPanel, String pos)
    {
        super(pos);
        attachedPanel = _attachedPanel;
        //TODO: write new listener
        //addActionListener(new PositionMenuItemListener(tc, attachedPanel));
    }
}
```

Now go into `makeMenus` and change

```
positionMenu.add(new PositionMenuItem(this, leftPanel, "left"));
positionMenu.add(new PositionMenuItem(this, middlePanel, "middle"));
positionMenu.add(new PositionMenuItem(this, rightPanel, "right"));
```

to

```
positionMenu.add(new PositionMenuItem(leftPanel, "left"));
positionMenu.add(new PositionMenuItem(middlePanel, "middle"));
positionMenu.add(new PositionMenuItem(rightPanel, "right"));
```

Once you do this, the program will compile. You no longer need either external class controlling the position menu. Close them.

Next we write the listener as an anonymous inner class. Begin by making the shell.

```
class PositionMenuItem extends JMenuItem
{
    public final ColorPanel attachedPanel;
    public PositionMenuItem(ColorPanel _attachedPanel, String pos)
    {
        super(pos);
        attachedPanel = _attachedPanel;
        addActionListener(new ActionListener(){
```

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

What needs to happen when a new position is selected? We just change the current panel. No graphical change is needed. So let's add the code.

```
class PositionMenuItem extends JMenuItem
{
    public final ColorPanel attachedPanel;
    public PositionMenuItem(ColorPanel _attachedPanel, String pos)
    {
        super(pos);
        attachedPanel = _attachedPanel;
        addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                currentPanel = attachedPanel;
            }
        });
    }
}
```

6 Cruft Patrol!

It seems some getter and setter methods in `Tricolor` are obviated. What can we trim out? Comment these out.

```
public ColorPanel getCurrentPanel()
{
    return currentPanel;
}
public void setCurrentPanel(ColorPanel c)
{
    currentPanel = c;
}
```

Everything still works, so you can get rid of them.

7 The Product

We are left with two classes. Here is `ColorPanel.java`, which we never changed

```
import javax.swing.JPanel;
import java.awt.Color;
public class ColorPanel extends JPanel
{
    private Color color;
    public ColorPanel()
    {
        color = Color.white;
    }
    public void setColor(Color _color)
    {
        color = _color;
    }
    public Color getColor()
    {
        return color;
    }
}
```

Here we show the newly slimmed-down `Tricolor` class in its entirety.

```
import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;


public class Tricolor extends JFrame implements Runnable
{
    ColorPanel leftPanel;
    ColorPanel rightPanel;
    ColorPanel middlePanel;
    ColorPanel currentPanel;

    public Tricolor()
```

```
{
    super("Tricolor");
    leftPanel = new ColorPanel();
    middlePanel = new ColorPanel();
    rightPanel = new ColorPanel();
    currentPanel = leftPanel;
}
public void run()
{
    setSize(500,500);
    makeMenus();
    //install Color Panels
    Container c = getContentPane();
    c.setLayout(new GridLayout(1,3));
    c.add(leftPanel);
    c.add(middlePanel);
    c.add(rightPanel);
    setVisible(true);
}
private void makeMenus()
{
    //make and install menu bar
    JMenuBar mbar = new JMenuBar();
    setJMenuBar(mbar);
    //File menu
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
    JMenuItem quitItem = new JMenuItem("Quit");
    fileMenu.add(quitItem);
    quitItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            System.exit(0);
        }
    });
    // Color Menu
    JMenu colorMenu = new JMenu("Color");
    mbar.add(colorMenu);
    colorMenu.add(new ColorMenuItem(Color.RED, "red"));
    colorMenu.add(new ColorMenuItem(Color.GREEN, "green"));
    colorMenu.add(new ColorMenuItem(Color.BLUE, "blue"));
    JMenu positionMenu = new JMenu("Position");
    // Position Menu
    mbar.add(positionMenu);
    positionMenu.add(new PositionMenuItem(leftPanel, "left"));
    positionMenu.add(new PositionMenuItem(middlePanel, "middle"));
```

```
        positionMenu.add(new PositionMenuItem(rightPanel, "right"));
    }
    class ColorMenuItem extends JMenuItem
    {
        private final Color color;
        public ColorMenuItem(Color _color, String _colorName)
        {
            super(_colorName);
            color = _color;
            addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e)
                {
                    currentPanel.setColor(color);
                    currentPanel.repaint();
                }
            });
        }
    }

    class PositionMenuItem extends JMenuItem
    {
        public final ColorPanel attachedPanel;
        public PositionMenuItem(ColorPanel _attachedPanel, String pos)
        {
            super(pos);
            attachedPanel = _attachedPanel;
            addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e)
                {
                    currentPanel = attachedPanel;
                }
            });
        }
    }

    public static void main(String[] args)
    {
        Tricolor t = new Tricolor();
        javax.swing.SwingUtilities.invokeLater(t);
    }
}
```

Programming Exercises

1. Add a new item to the color menu that sets the current panel to white.

2. Add a new panel and modify the position menu to accommodate it.

8 Inner Classes in General

We have inner classes in several guises during the study of the tricolor application. Let us now summarize and consolidate what we have seen.

Inner classes can be static. Such classes have access to private methods and variables of the enclosing class, but only via an instance of that class. In the code, shown here, note the illegal and legal examples.

```
public class Outer
{
    int x;
    int y;
    public Outer(... )
    {
        //constructor code
    }
    private foo(...)
    {
        //code
    }
    static class Inner
    {
        public void go()
        {
            x = 5  //Illegal
            Outer o  new Outer(...);
            o.x = 5 //OK
            foo(....);  // Illegal
            o.foo(...)  //OK
        }
    }
}
```

Remember, non-static portions of a class have access to static portions, but direct access (not via an instance) is not allowed. If you are declaring an inner class `static` you should closely consider making it a separate class, unless it relieves a lot of complexity.

However, making an inner class static defeats the purpose of its inner-ness: access to the enclosing class's state variables. If you have a class that you would like to make innner and static, here is a better way to deal with it.

```
public class Outer
{
    int x;
    int y;
    public Outer(... )
    {
        //constructor code
    }
    private foo(...)
    {
        //code
    }
}
static class Inner
{
    public void go()
    {
        x = 5  //Illegal
        Outer o  new Outer(...);
        o.x = 5 //OK
        foo(....);  // Illegal
        o.foo(...)  //OK
    }
}
```

All you are doing here is making it a non-public class within the same file. Such a class is not visible outside of the file.

This is in contrast to the non-static inner classes we used to control the menus. These were created so as to have access to the state variables of the enclosing outer class. We created the named inner classes `PositionMenuItem` and `ColorMenuItem`.

The anonymous class we created as an event handler for `quitItem` is an example of a *local class*, since it is created inside of a method of the enclosing class. Local classes do not have access to the local variables of their enclosing method, unless the variable is declared `final`. Accessing a local non-final variable is a compiler error.

We will return to topic of inner classes when we discuss Java collections; inner classes can play an important role in the creation of data structures.

9 Adding and Deleting Components from a `JFrame`

We will show a program that adds and removes buttons from a window "on the fly." Begin by creating a graphical shell.

```
import javax.swing.JFrame;

public class Adder extends JFrame implements Runnable
{
    public Adder()
    {
        super("Add and Remove Buttons Demo");
    }
    public void run()
    {
        setSize(600,600);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        Adder a = new Adder();
        javax.swing.SwingUtilities.invokeLater(a);
    }
}
```

Run and compile this; the result is an empty window with a title in the title bar specified by the constructor.

Next, on the top of the window, we will add buttons named `"Add"` and `"Remove"`. To do this we will take take the following steps.

1. Make a JPanel with a 1 row 2 column grid layout.
2. Add the two buttons to it
3. Add it to the north side of the content pane using the `BorderLayout` static constant `NORTH`.

We modify the `run()` method as follows.

```
public void run()
{
    setSize(600,600);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JPanel topPanel = new JPanel(new GridLayout(1,2));
    JButton addButton = new JButton("Add");
    topPanel.add(addButton);
    JButton removeButton = new JButton("Remove");
    topPanel.add(removeButton);
    getContentPane().add(BorderLayout.NORTH, topPanel);
    setVisible(true);
}
```

Do not neglect to add these imports.

```
import javax.swing.JButton;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.GridLayout;
```

You will see that the add and remove buttons now appear at the top of the content pane.

Next, we shall make an inner class that extends `JPanel` and which overrides its `paintCompnent` method. We will put at 10×10 grid layout in this panel.

Here it is. Do not forget to import `java.awt.Graphics`.

```
class ButtonPanel extends JPanel
{
    public ButtonPanel()
    {
        super(new GridLayout(10,10));
    }
    @Override
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
    }
}
```

This is the panel in which we will keep our buttons. Let's make an array list of buttons and make it a state variable. We will also make the button panel a state variable and add it to the window. We will initialize it in the constructor. Add this to your the top of the class

```
private ArrayList<JButton> buttons;
private ButtonPanel bp;
```

Inside the constructor add

```
buttons = new ArrayList<JButton>();
bp = new ButtonPanel();
```

To your imports add

```
import java.util.ArrayList;
```

Now let us add code to cause pushing the Add button to addd buttons to the array list `buttons`. We do this by creating an action listener and attaching it to `addButton` in the `run` method as follows. Note the use of the if statement to prevent overpopulation.

```
addButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        int n = buttons.size();
        if(n < 100)
        {
             buttons.add(new JButton("" + buttons.size()));
             repaint();
        }
    }
});
```

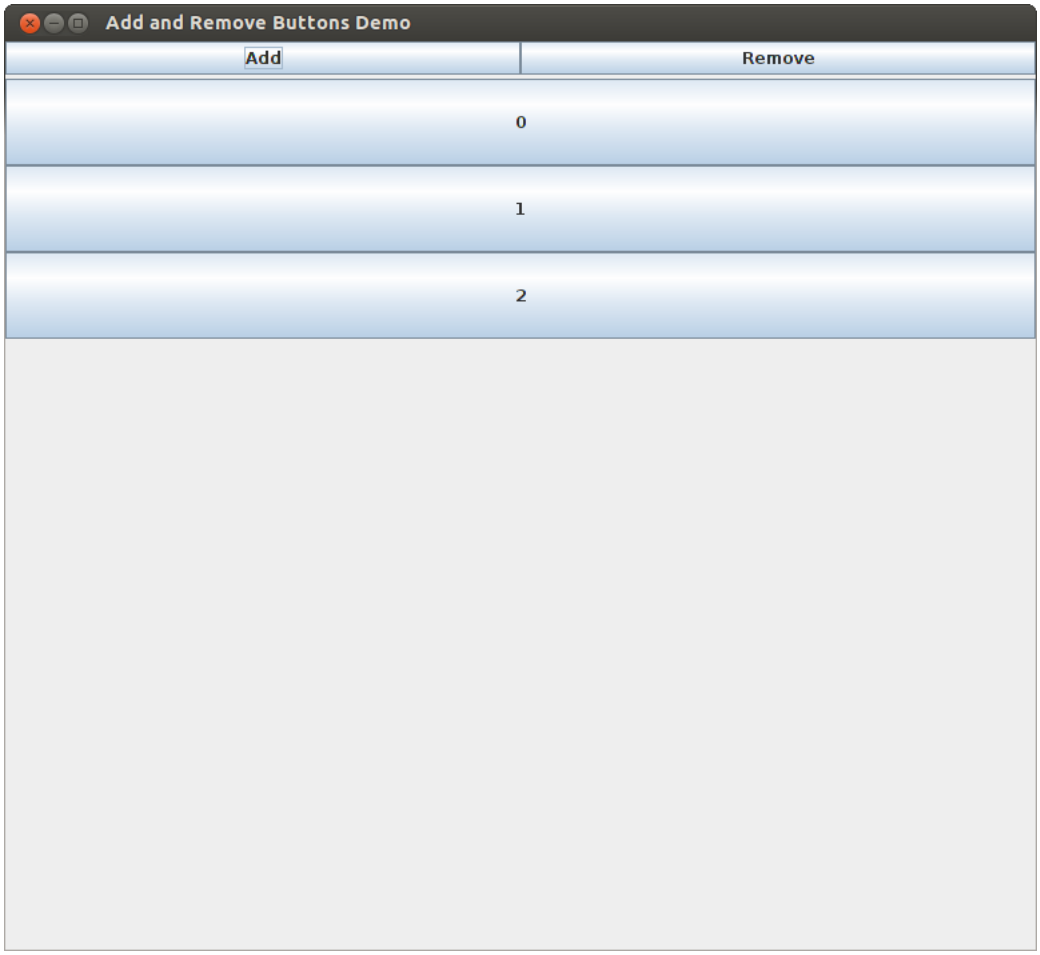
If you hit the add button, you will not see any buttons added to the screen. Stick in this line of code to see the listener is working.

```
System.out.printf("buttons.size() = %s\n", buttons.size());
```

Hit the add button and see this in the console.

```
buttons.size() = 1
buttons.size() = 2
buttons.size() = 3
buttons.size() = 4
buttons.size() = 5
buttons.size() = 6
```

The array list is being populated. However, we are not seeing the buttons appear on in the lower part of the console. Let us get them in using the `repaint()` method. Before proceeding, delete the print line you just inserted. Now we are going to test this out. Run the app and hit the add button three times. You will see nothing has happened. Now maximize or resize the window. That triggers a repaint. You will see this.



Now hit the add button five more times. You will see no changes. Then trigger a repaint by resizing or maximizing. You will then see what you expect, which is this.

The window does not behave as expected. You have to trigger a repaint to see the added buttons. We don't want this. Notice that containers are smart, if you add the same widget several times, it will only appear once.

Next, let us try to remove the last button we put in. Add this listener. Note the use of the if statement to prevent a "blood from a turnip" situation that could cause a nasty exception.

```
removeButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        int n = buttons.size();
        if(n > 0)
        {
            buttons.remove(n - 1);
```

```
                repaint();
            }
        }
});
```

Now run the app and click the add button five times. Resize or maximize the window. The five buttons then suddenly appear. Now click the remove button once and resize. You will see this.

It is whacko and it is not what we expect. What is the remedy for resetting the components in a `JPanel`? Modify your `paintComponent` as follows.

```
@Override
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    removeAll();
    for(JButton b: buttons)
    {
```

```
            add(b);
        }
        revalidate();
    }
```

This evicts all items from the window, adds back in the ones that belong, and then `revalidate()` ensures that the components themselves refresh. This goes deeper than just repainting. Your add and remove features now work as expected. Now keep clicking until the whole window fills up with buttons.