Two cool ideas:

Anonymous classes

Polymorphism

Anonymous classes – motivation

- You probably have many buttons and/or menu items in your VectorGraphics project
- Three approaches for responding to the events from selecting those buttons / menu items
  1. Classic: Each button is a class that implements ActionListener
  2. Least code: Panel responds to ALL buttons
  3. ***Anonymous class*** for each button

Anonymous classes – motivation

1. Classic: Each button (likewise for menu-item) is a class that implements ActionListener
   - Obeys Quality Tip: Buttons should respond to themselves

```
public class XXXButton extends JButton
                       implements ActionListener {

    public XXXButton(XXPanel panel) {
        // store panel in field
    }

    public void actionPerformed(ActionEvent event) {
        // Ask panel to ...
    }
```

Anonymous classes – motivation

2. Panel responds to ALL the buttons and menu-items
   - Not very OO, but easy to code

Or **this** if this code is in the Panel class

```
Wherever buttons are constructed:
   button.addActionListener(panel);
```

```
public class XXXPanel extends JPanel
                    implements ActionListener {

    public void actionPerformed(ActionEvent event) {
        JButton button = (JButton) (event.getSource());
        if (button.getText().equals("Make rectangle") {
            // construct and draw a rectangle
        } else if (...) {
            // etc
        }  // etc
    }
}
```

Anonymous classes

3. Button responds via an ***anonymous class***
   - Responding code is physically close to constructing code
   - Code in red below is the anonymous class

```
Wherever buttons are constructed:

button.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent event) {
        // Ask panel to ...
    }
});
```

The anonymous class is an ***inner class*** and hence can refer to:

- Any field of the enclosing class
- Any local variable in the enclosing method if the variable is *final*.

Polymorphism

- You probably have a list of objects that paintComponent draws:

```
ArrayList<MyShape> objectsToDraw;
```

- Suppose MyShape is an interface that specifies a draw method that takes a Graphics object. Then *paintComponent(Graphics g)* can be:

```
for (MyShape objectToDraw : objectsToDraw) {
    objectToDraw.draw(g);
}
```

- At run time, each *objectToDraw* ***morphs*** into the particular type it ***actually*** is, and uses its ***actual*** *draw* method.
- Bottom-line: for any statement like

```
x.foo(…);
```

the ***actual*** type of *x* (not the declared type) is what determines which *foo* function to run