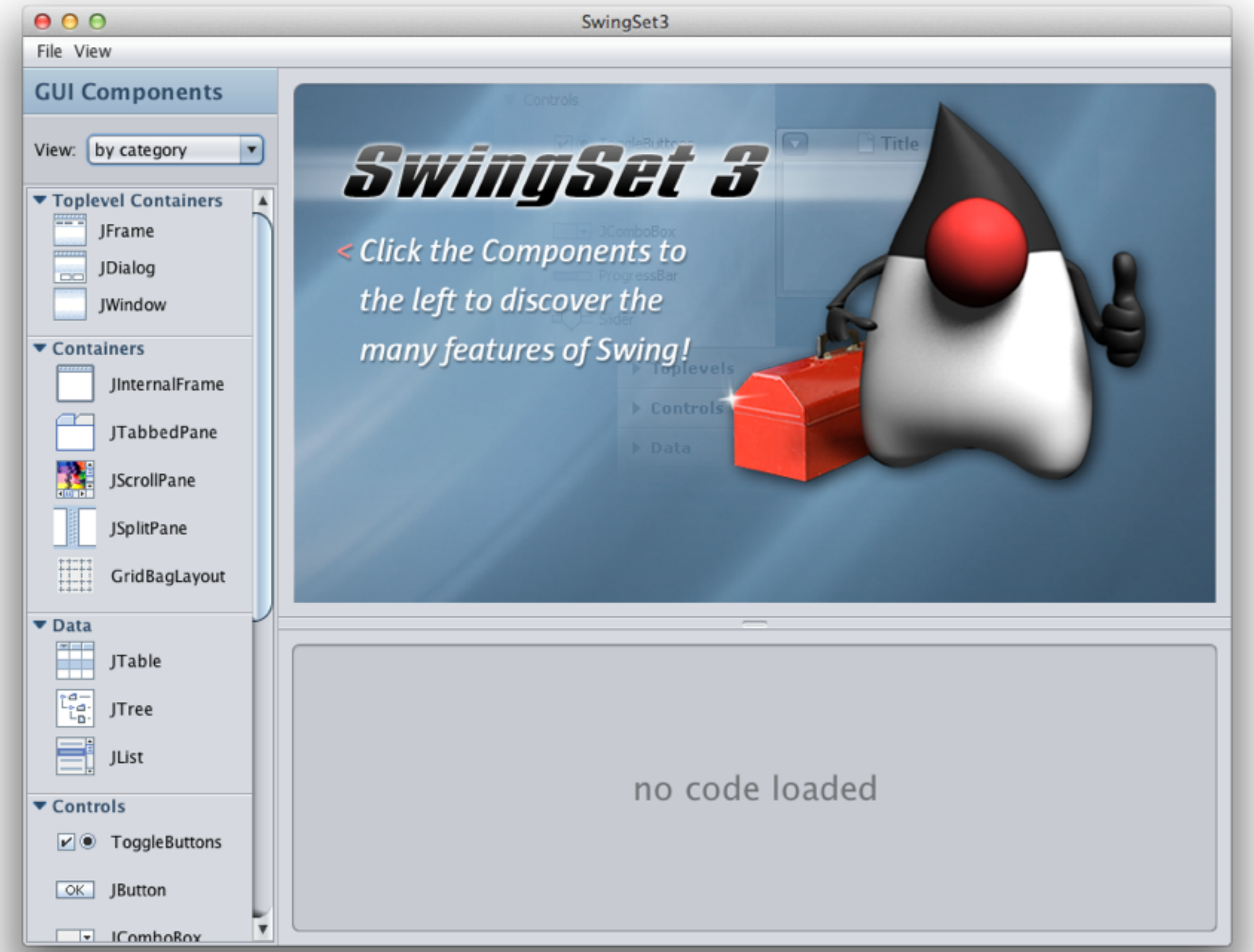
COM1003 Java Programming

Richard Clayton



A Very Graphic Story

Part 2 – Graphics and Graphical User Interfaces in Java

Making GUIs do things

So far we’ve laid out GUI components onto a container, but the GUI doesn’t actually *do* anything yet

We need to get GUI components to respond to user button clicks, mouse drags, window focussing etc.

The Event Model

GUI components in Java generate objects when things happen to them; e.g mouse clicks, resize, mouse drag.

These objects are messages called Events.

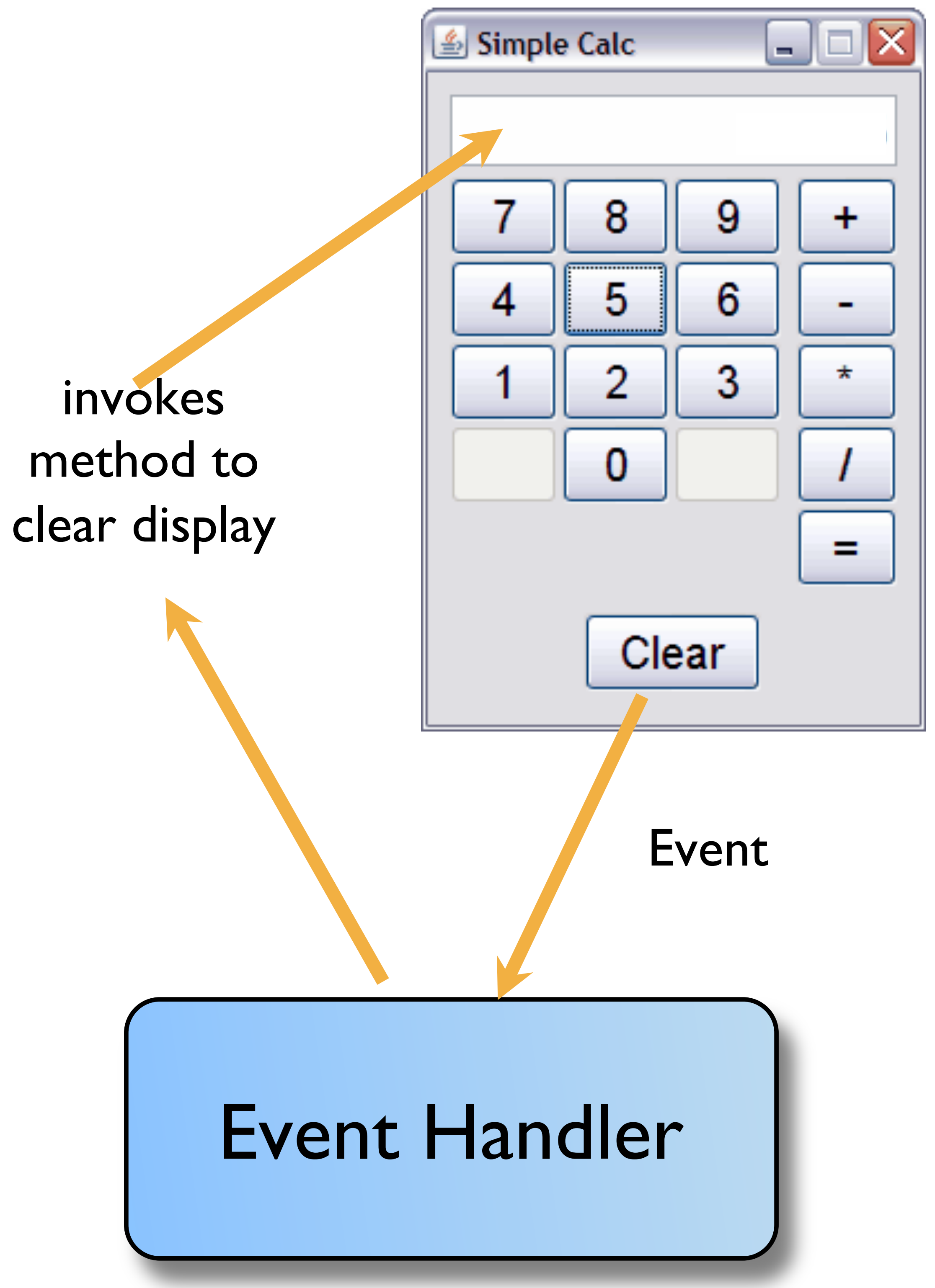
- Event objects encapsulate precise information about the event, for example the (x, y) co-ordinates of the button click.

We then need to attach special Event Listener objects to GUI components.

- Events generated by the GUI components are sent to the attached Event Listeners.
- We write Event Listeners to decide what should occur when an event has happened (e.g. perform some action).

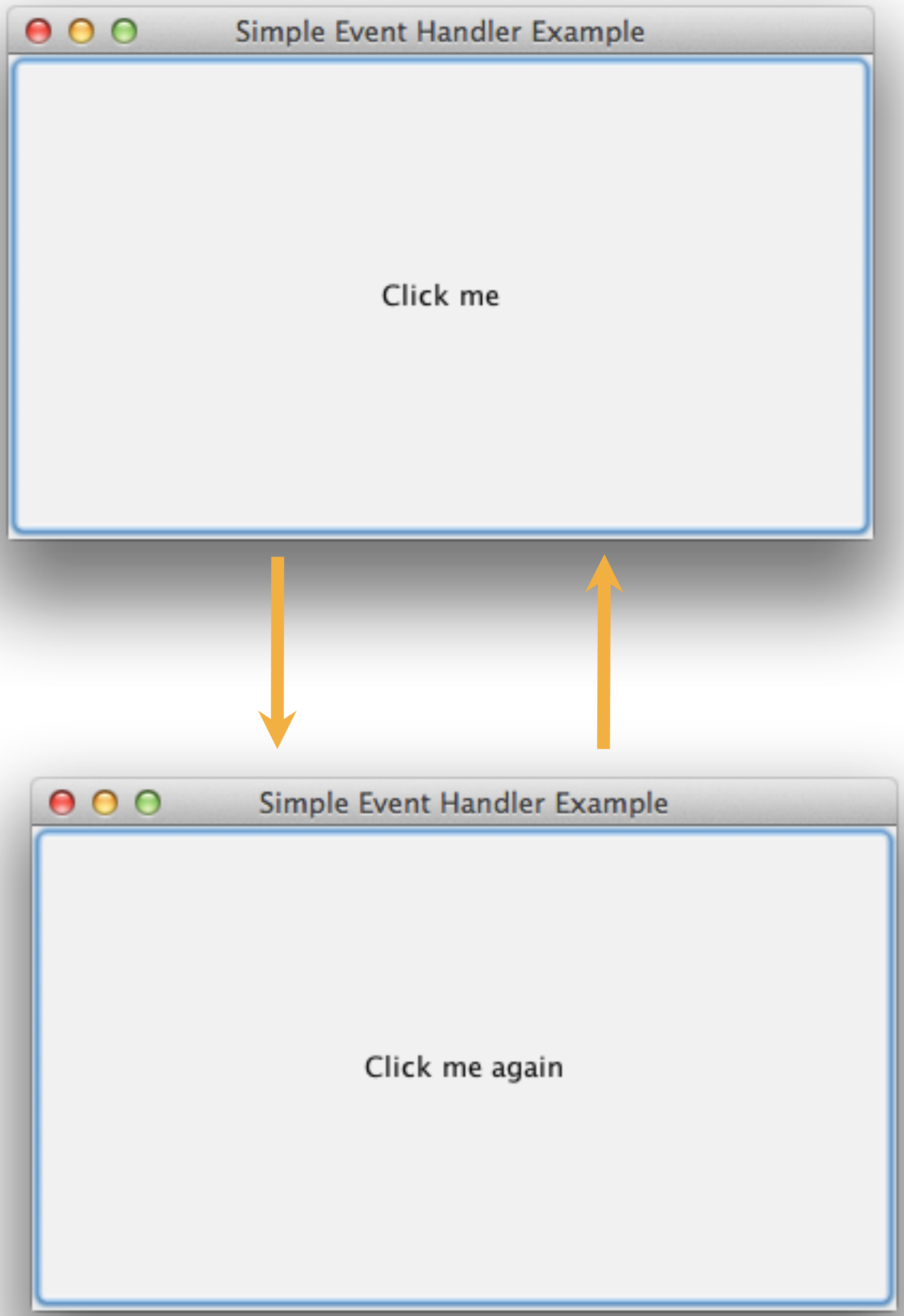
Example

1. User clicks "Clear" button
2. The clear button GUI component generates an Event.
3. The event is sent to the Event Listener
4. The Event Listener clears the drawing canvas



Worked Example

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;

public class SimpleEventHandlerExample extends CentredFrame
                                     implements ActionListener {

    JButton button;
    String[] labels = {"Click me", "Click me again"};
    int currentLabel = 0;

    public SimpleEventHandlerExample() {
        button = new JButton(labels[0]);
        button.addActionListener(this);

        setTitle("Simple Event Handler Example");
        getContentPane().add(button);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        currentLabel ++;
        if (currentLabel >= labels.length) {
            currentLabel = 0;
        }
        button.setText(labels[currentLabel]);
    }

    public static void main(String[] args) {
        new SimpleEventHandlerExample();
    }
}
```

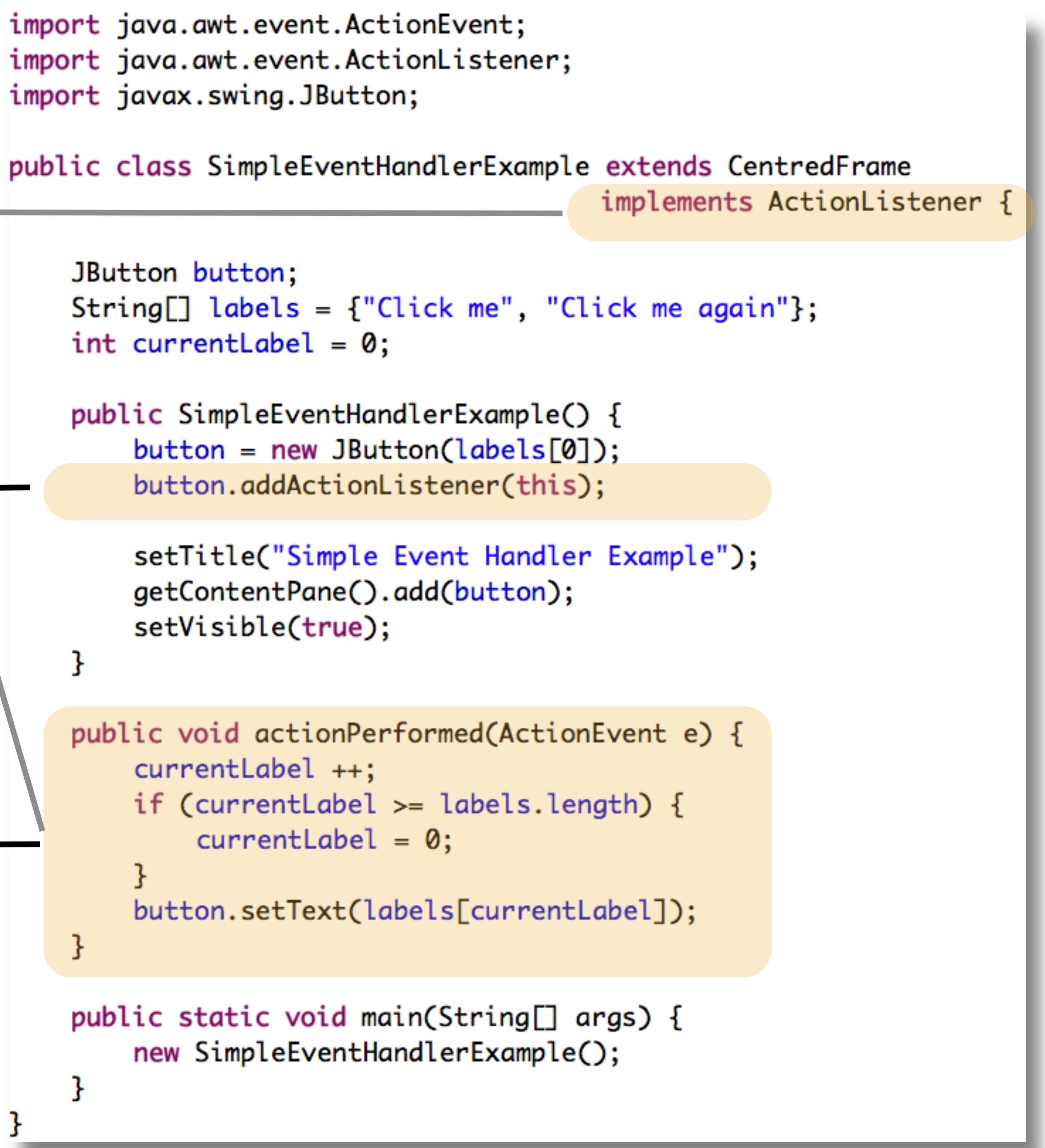


JButton and Event Handling

Our class implements ActionListener, the Event Handler for buttons. One method must be implemented, actionPerformed.

We need to register the Event Handler with our button. This tells the button to send the ActionEvent objects it generates on button clicks to our ActionListener, so that our interface can respond appropriately.

The actionPerformed method receives the ActionEvent objects generated by the button clicks. The ActionEvent objects themselves are not so interesting here - more the fact that the event has actually occurred. The method responds to the event by changing the label of the button.



```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;

public class SimpleEventHandlerExample extends CentredFrame
        implements ActionListener {

    JButton button;
    String[] labels = {"Click me", "Click me again"};
    int currentLabel = 0;

    public SimpleEventHandlerExample() {
        button = new JButton(labels[0]);
        button.addActionListener(this);

        setTitle("Simple Event Handler Example");
        getContentPane().add(button);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        currentLabel ++;
        if (currentLabel >= labels.length) {
            currentLabel = 0;
        }
        button.setText(labels[currentLabel]);
    }

    public static void main(String[] args) {
        new SimpleEventHandlerExample();
    }
}
```

Where should the Event Handler code go?

In the last example, the frame implemented the listener.

- But in general there may be more than one button, and we need different event handling code for each button.
- Or, there may be different types of GUI component with different types of listener
- Ideally different handling code should be kept separate and put into different classes

Three possible different approaches. Put the handler in:

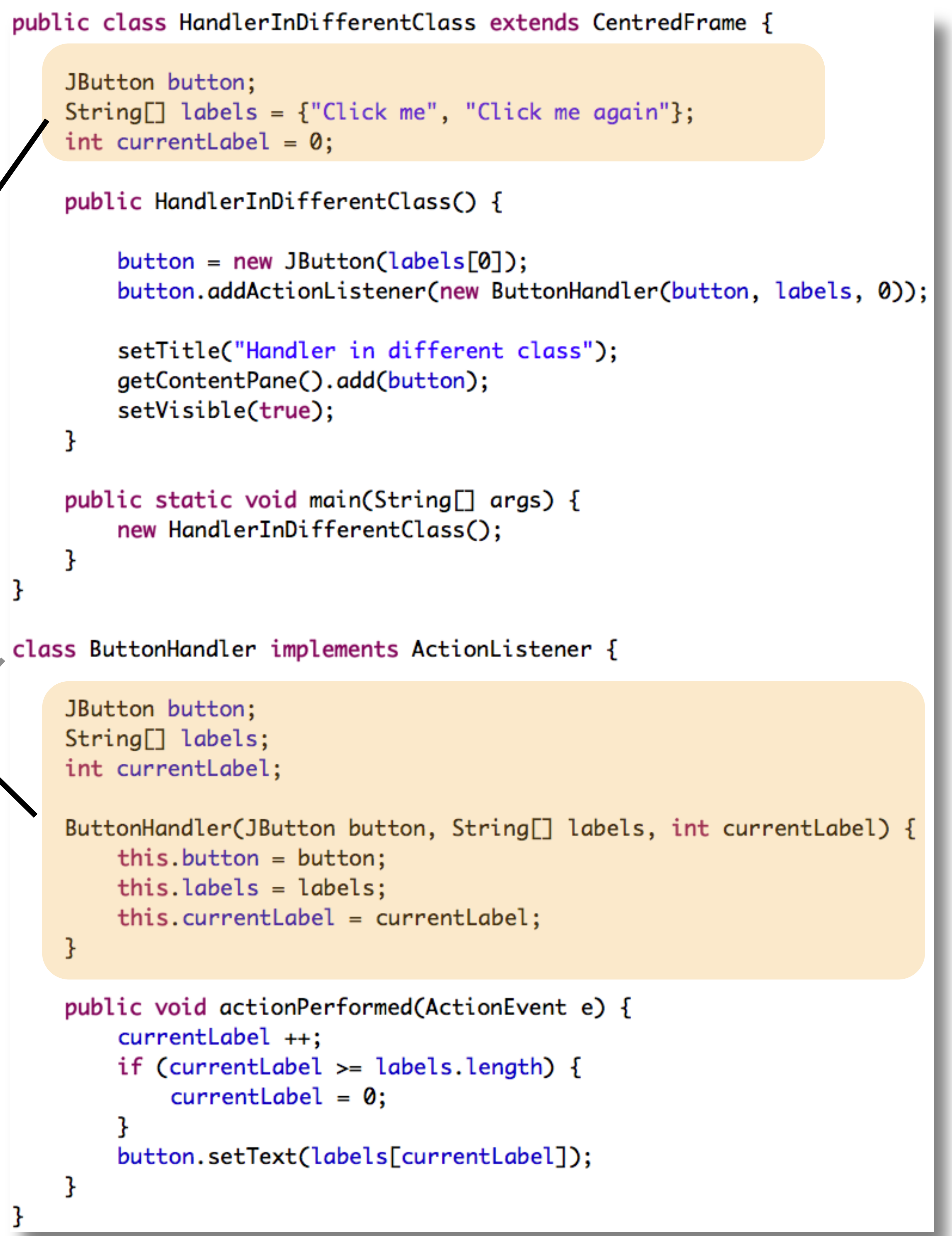
1. Different class
2. Non-static member class
3. Anonymous class

See example code in com1003.guis.eventhandling package

1. Handler in a different class

The handler code is now in a separate class but a lot of duplicate reference variables have to be created

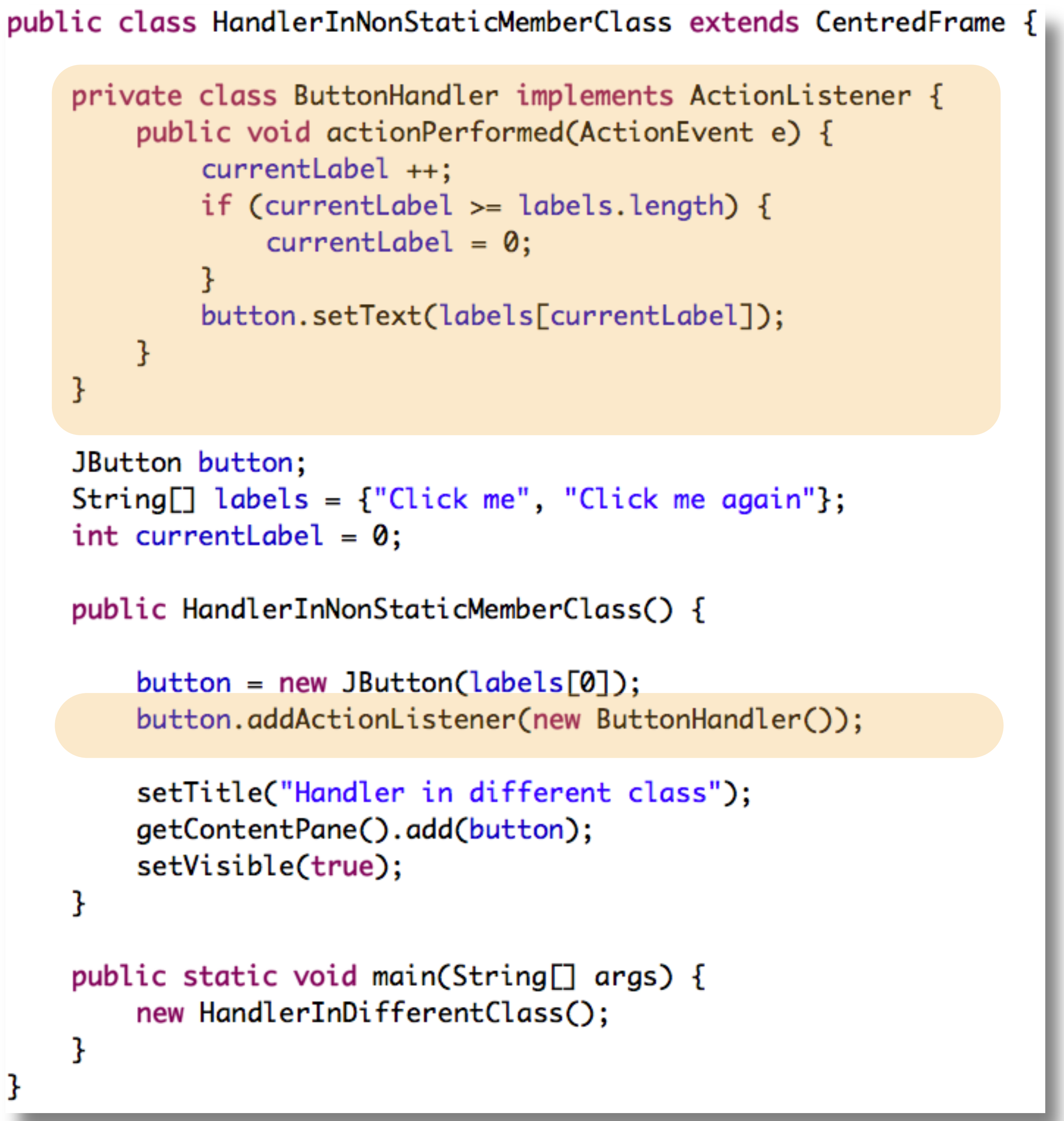
Also the class is visible to the rest of the package



```java
public class HandlerInDifferentClass extends CentredFrame {

    JButton button;
    String[] labels = {"Click me", "Click me again"};
    int currentLabel = 0;

    public HandlerInDifferentClass() {

        button = new JButton(labels[0]);
        button.addActionListener(new ButtonHandler(button, labels, 0));

        setTitle("Handler in different class");
        getContentPane().add(button);
        setVisible(true);
    }

    public static void main(String[] args) {
        new HandlerInDifferentClass();
    }
}

class ButtonHandler implements ActionListener {

    JButton button;
    String[] labels;
    int currentLabel;

    ButtonHandler(JButton button, String[] labels, int currentLabel) {
        this.button = button;
        this.labels = labels;
        this.currentLabel = currentLabel;
    }

    public void actionPerformed(ActionEvent e) {
        currentLabel ++;
        if (currentLabel >= labels.length) {
            currentLabel = 0;
        }
        button.setText(labels[currentLabel]);
    }
}
```

2. Non-static member class

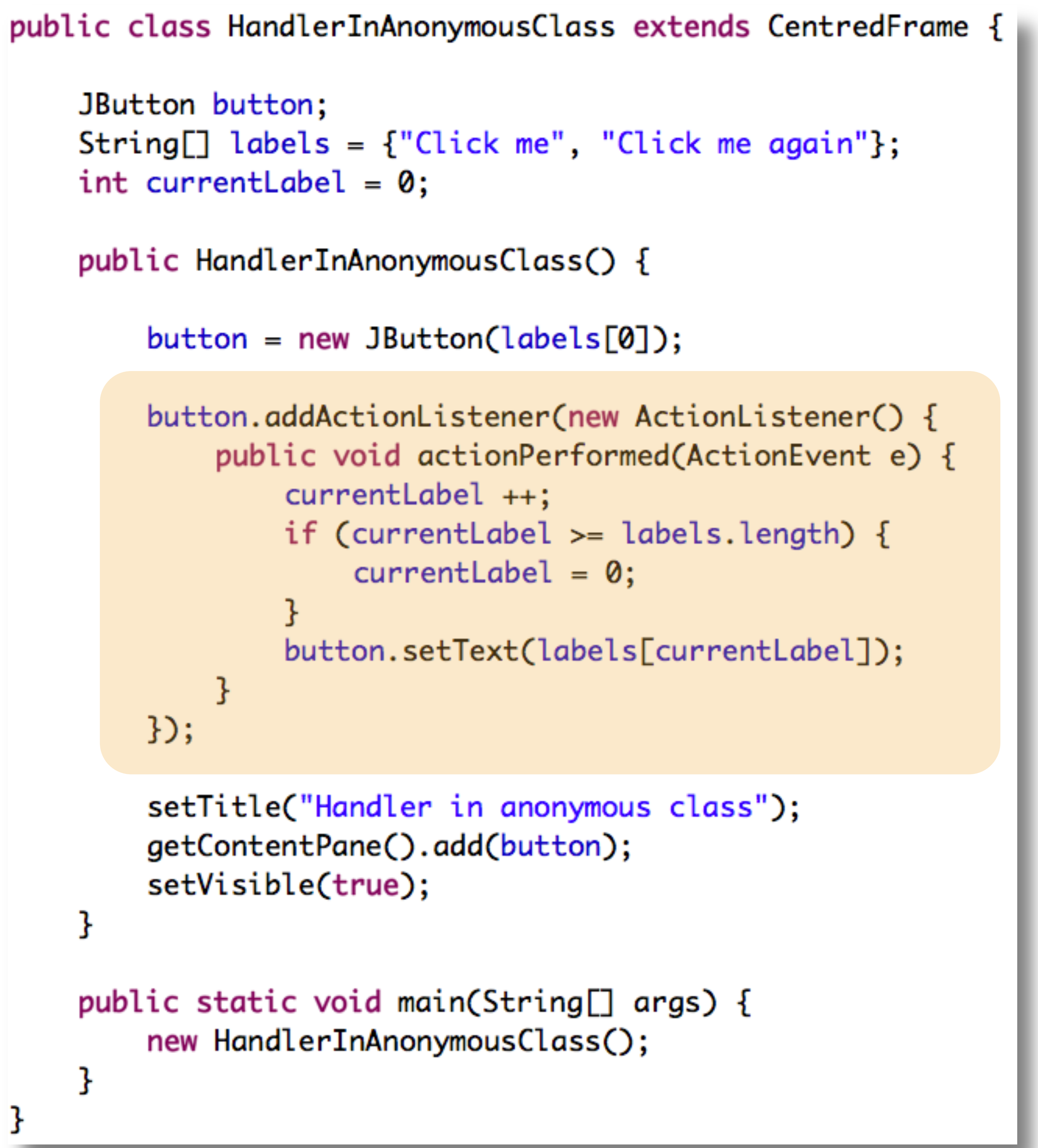
Non-static member classes have access to the instance variables of their outer class, so no need to duplicate reference variables.



```java
public class HandlerInNonStaticMemberClass extends CentredFrame {

    private class ButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            currentLabel ++;
            if (currentLabel >= labels.length) {
                currentLabel = 0;
            }
            button.setText(labels[currentLabel]);
        }
    }

    JButton button;
    String[] labels = {"Click me", "Click me again"};
    int currentLabel = 0;

    public HandlerInNonStaticMemberClass() {

        button = new JButton(labels[0]);
        button.addActionListener(new ButtonHandler());

        setTitle("Handler in different class");
        getContentPane().add(button);
        setVisible(true);
    }

    public static void main(String[] args) {
        new HandlerInDifferentClass();
    }
}
```

3. Anonymous Class

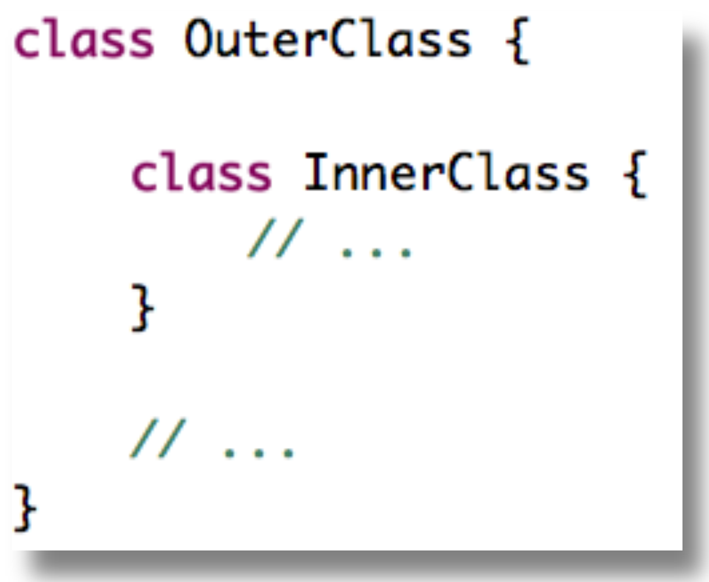
Less verbose



```java
public class HandlerInAnonymousClass extends CentredFrame {

    JButton button;
    String[] labels = {"Click me", "Click me again"};
    int currentLabel = 0;

    public HandlerInAnonymousClass() {

        button = new JButton(labels[0]);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                currentLabel ++;
                if (currentLabel >= labels.length) {
                    currentLabel = 0;
                }
                button.setText(labels[currentLabel]);
            }
        });

        setTitle("Handler in anonymous class");
        getContentPane().add(button);
        setVisible(true);
    }

    public static void main(String[] args) {
        new HandlerInAnonymousClass();
    }
}
```

Inner Classes

Java lets you define a class within another class. Such classes are called inner classes (or sometimes nested classes).



```
class OuterClass {

    class InnerClass {
        // ...
    }

    // ...
}
```

Graphical user interfaces (GUIs) tend to make extensive use of inner classes.

There is more we need to learn about classes before we can begin to implement GUIs with inner classes – so watch the inner classes video now.

Inner classes – a small diversion

Why have inner classes?

Logical grouping of classes

- If one class is only useful to one other class, then it’s a good idea to keep them together.

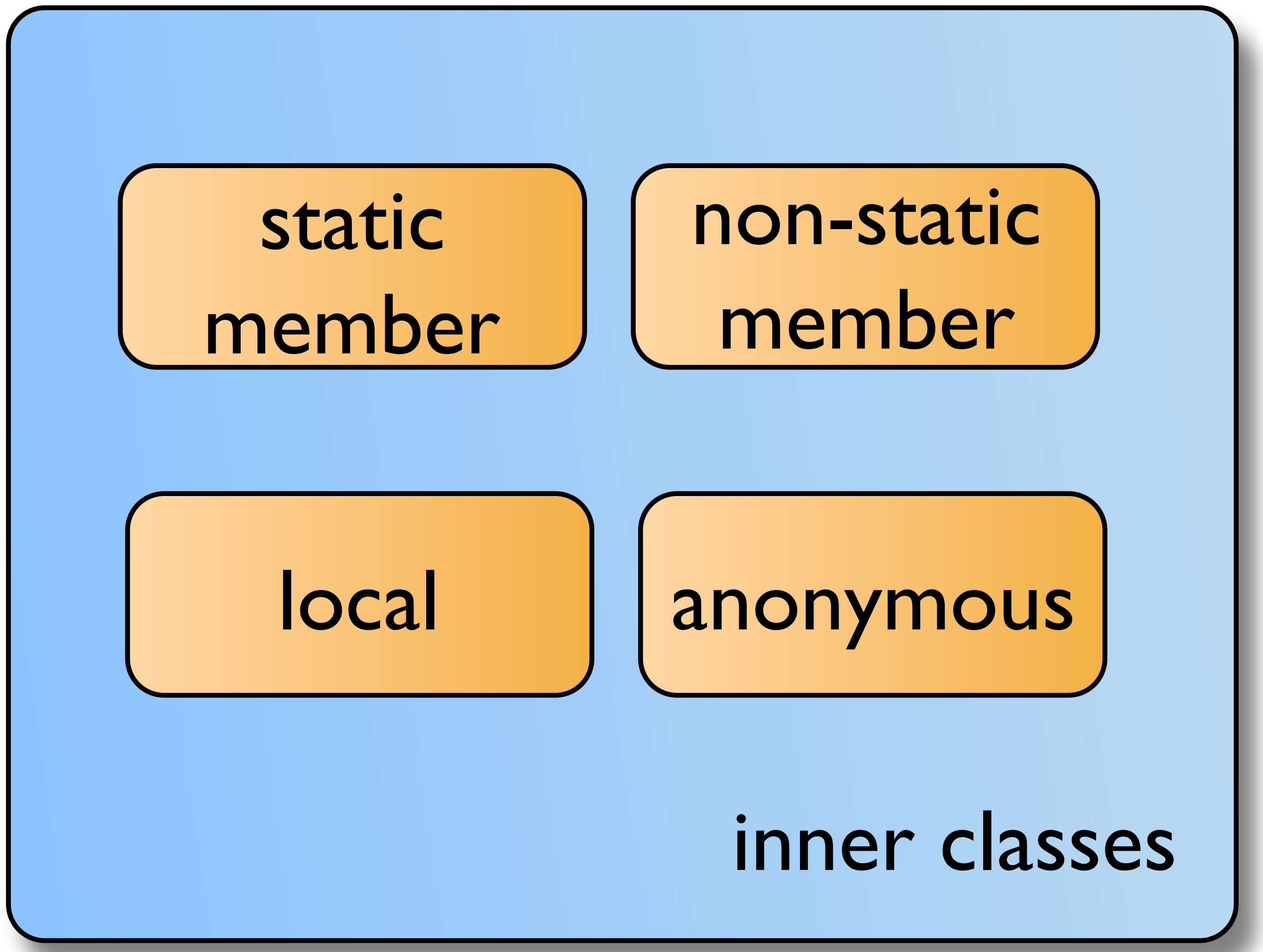
Increased Encapsulation

- The inner class can be hidden inside the outer class.

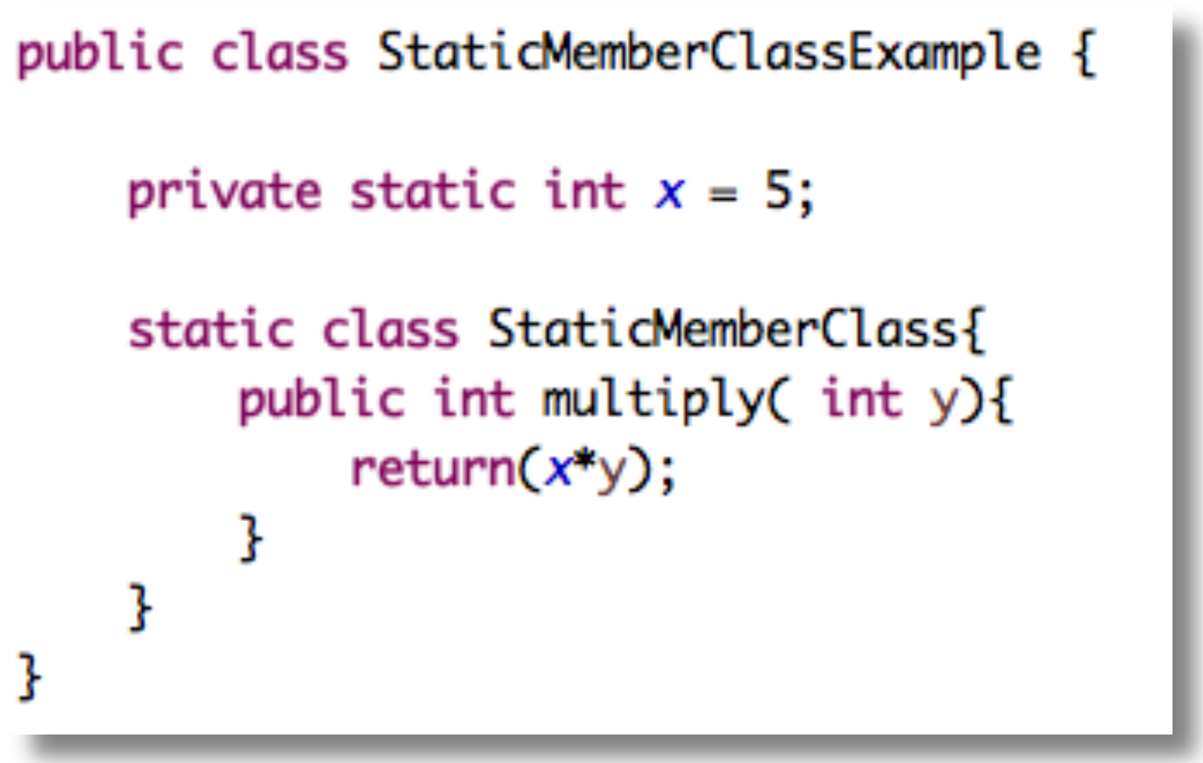
More readable, maintainable code

- Nesting smaller classes within another level classes places the code nearer to where it is used.

Four types of Inner Class



Static Member Classes

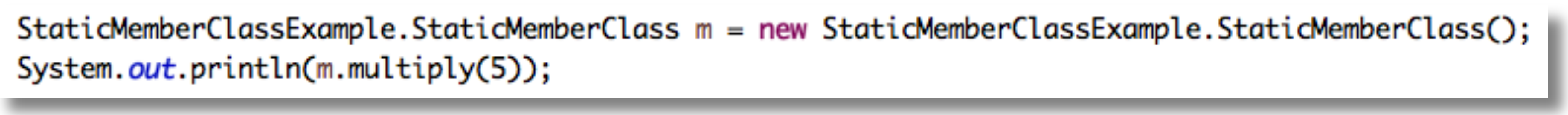


```
public class StaticMemberClassExample {

    private static int x = 5;

    static class StaticMemberClass{
        public int multiply( int y){
            return(x*y);
        }
    }
}
```

A static member class behaves like a top-level class that has been nested in another top-level class for packaging convenience.

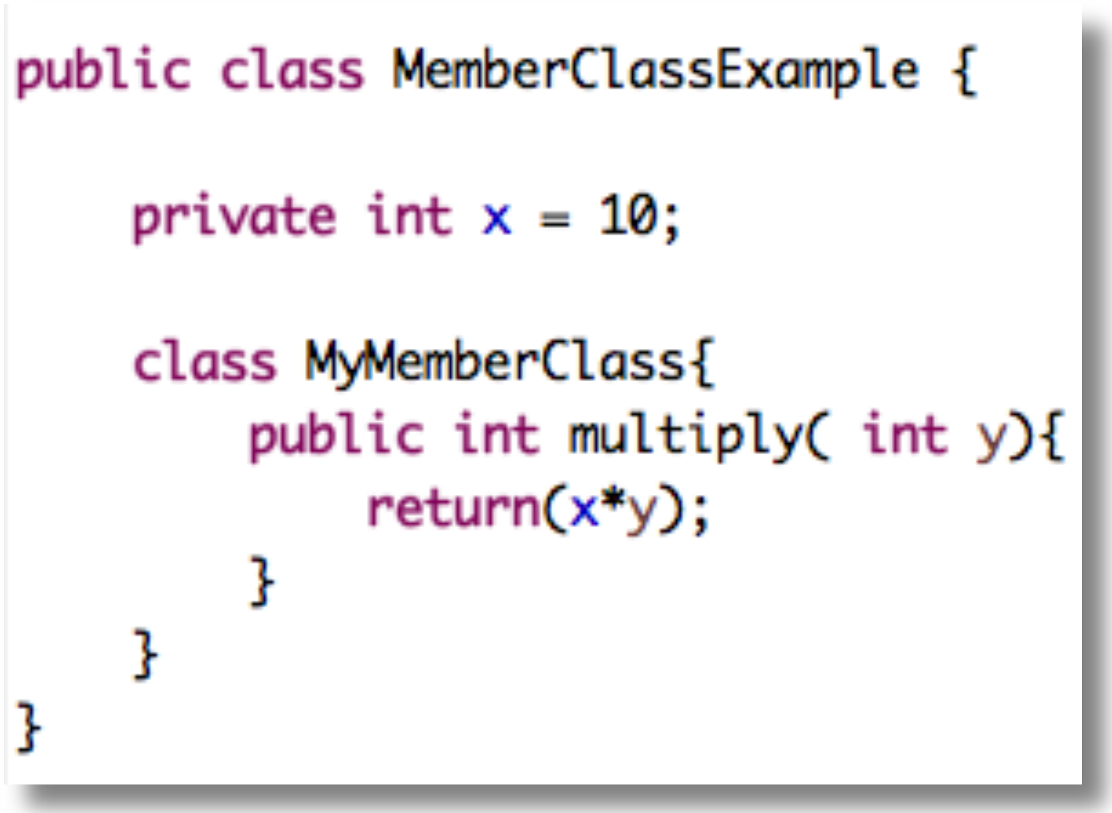
It can only access the static fields of the outer class, not instance fields.

From outside the outer class, static member classes can be accessed using the outer class name:



```
StaticMemberClassExample.StaticMemberClass m = new StaticMemberClassExample.StaticMemberClass();
System.out.println(m.multiply(5));
```

(Non-Static) Member Classes



```
public class MemberClassExample {

    private int x = 10;

    class MyMemberClass{
        public int multiply( int y){
            return(x*y);
        }
    }
}
```

A non-static member class is also defined as a member of an enclosing class, but is not declared with the static modifier.

It is analogous to an instance variable or method.

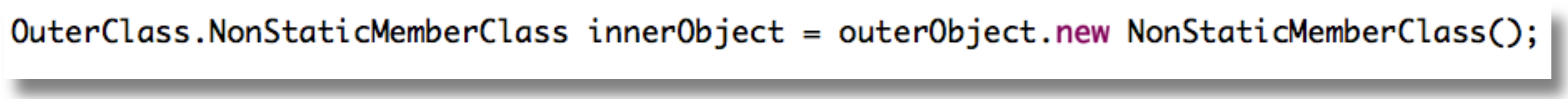
An instance of a member class is always associated with an instance of the outer class, and its code has access to all the fields and methods (static and non-static) of the outer class.

(Non-Static) Member Classes

An instance of a non-static member class can only exist within an instance of the outer class.

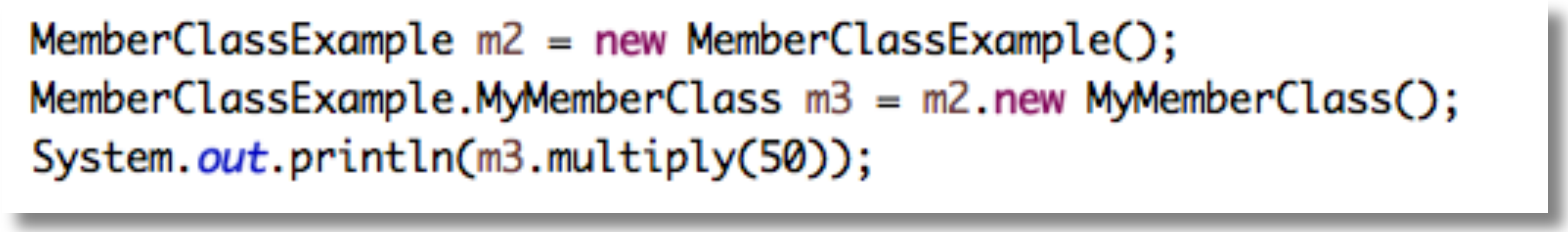
To instantiate the inner class, you must first instantiate the outer class.

From outside the outer class, the inner object can be created using the following syntax:



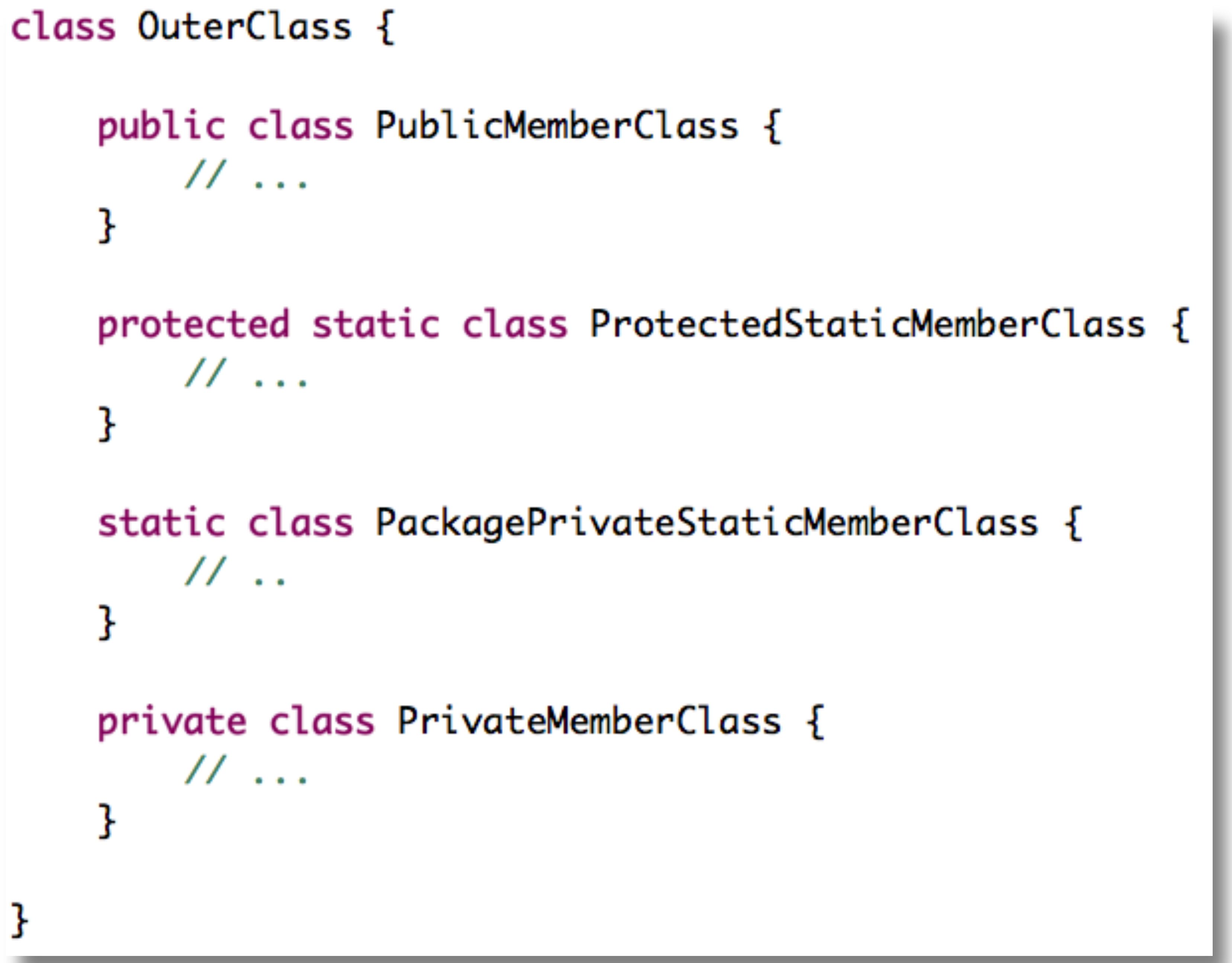
```
OuterClass.NonStaticMemberClass innerObject = outerObject.new NonStaticMemberClass();
```

So for the example on the previous slide we could use:



```
MemberClassExample m2 = new MemberClassExample();
MemberClassExample.MyMemberClass m3 = m2.new MyMemberClass();
System.out.println(m3.multiply(50));
```

Access Modifiers & Member Classes

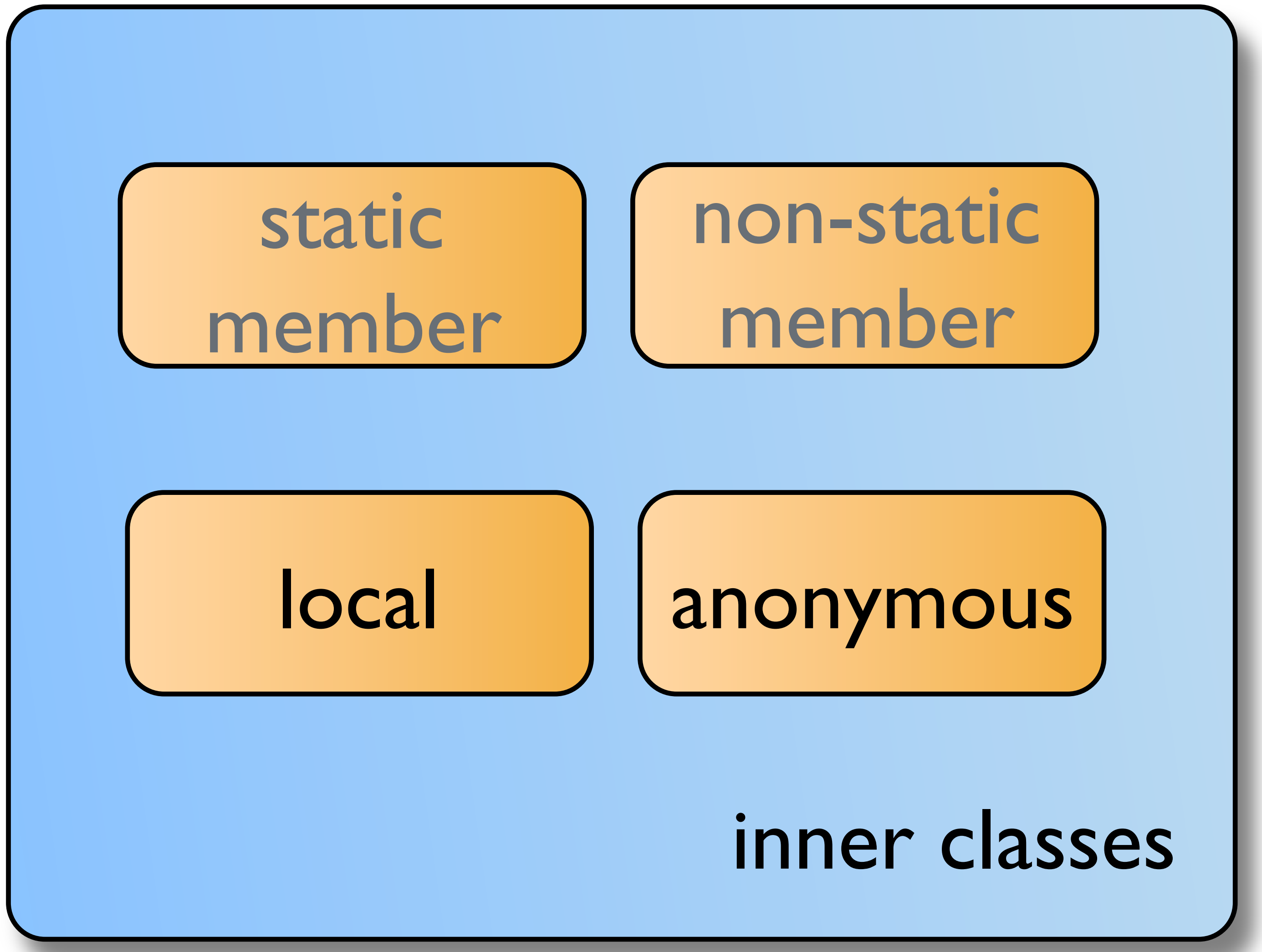


```
class OuterClass {

    public class PublicMemberClass {
        // ...
    }

    protected static class ProtectedStaticMemberClass {
        // ...
    }

    static class PackagePrivateStaticMemberClass {
        // ..
    }

    private class PrivateMemberClass {
        // ...
    }

}
```

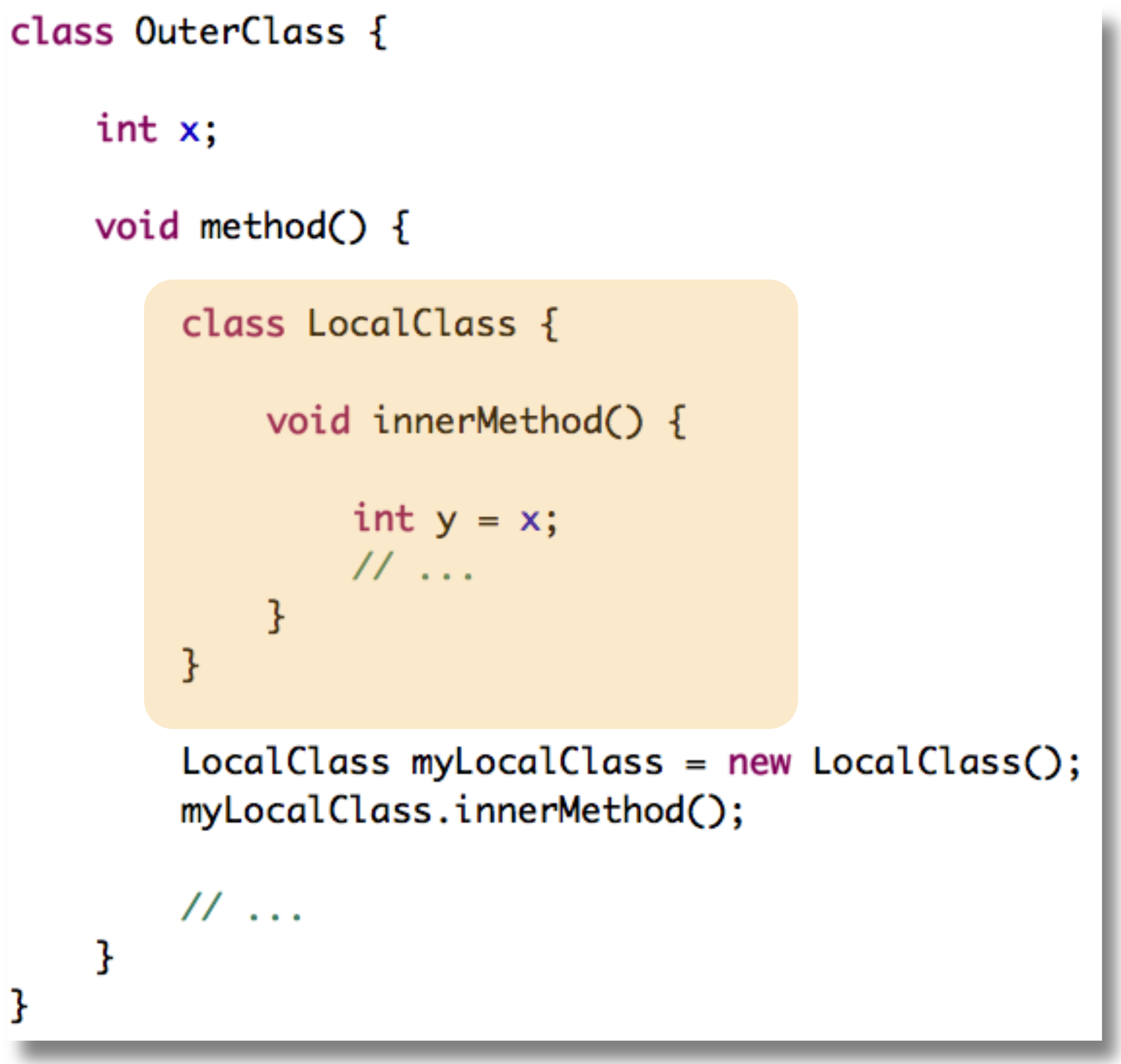
As members of the outer class, member classes (static and non-static) can be declared as public, protected, package-private or private.

If you are not sure about the implications … experiment!

Four types of Inner Class



Local Classes



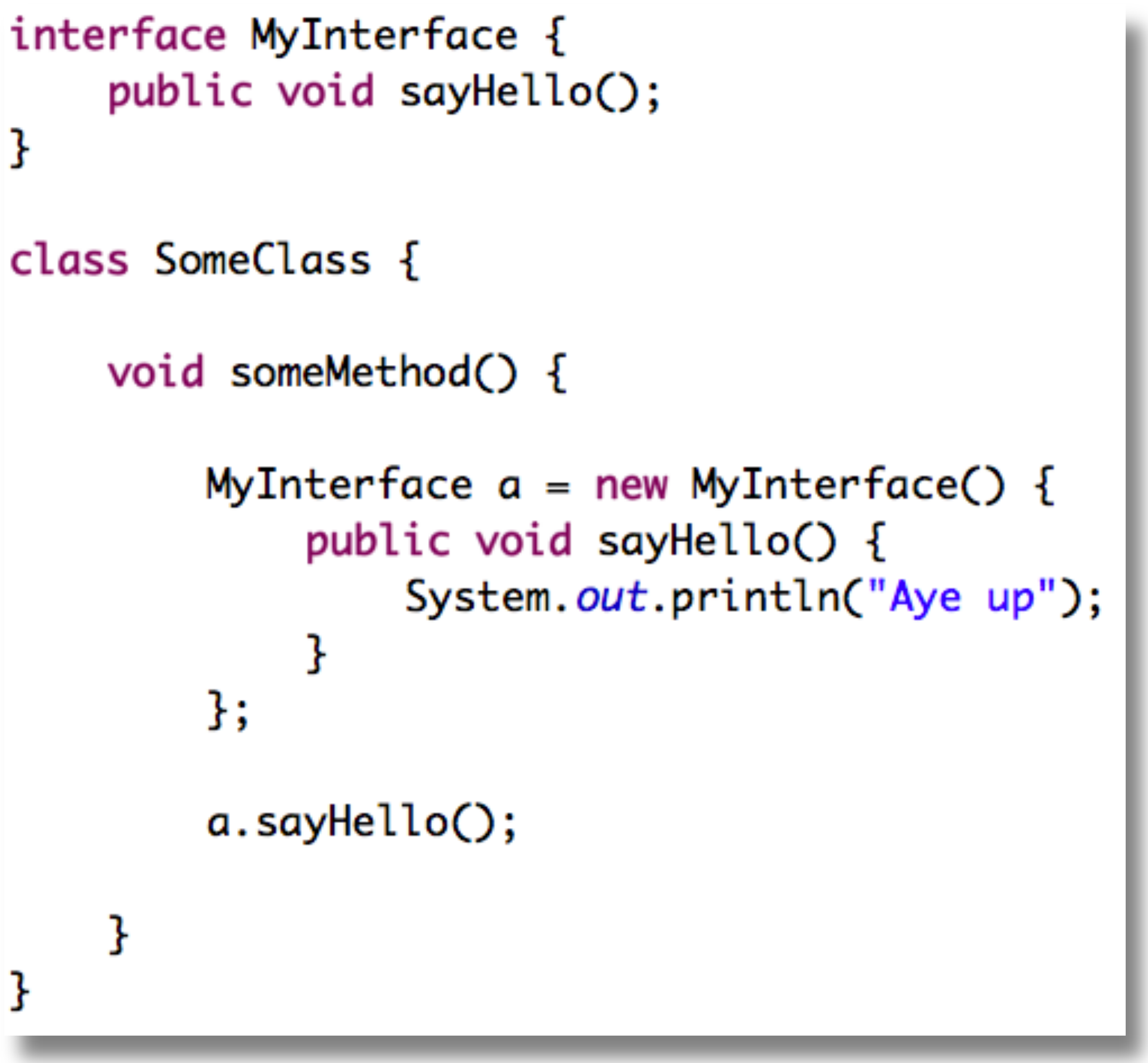
```
class OuterClass {

    int x;

    void method() {

        class LocalClass {

            void innerMethod() {

                int y = x;
                // ...
            }
        }

        LocalClass myLocalClass = new LocalClass();
        myLocalClass.innerMethod();

        // ...
    }
}
```

Local classes are similar to member classes, except they exist only in the block of code in which they are defined.

(As such there is no need to give them an access modifier).

Useful when a class is only used by one method.

Anonymous Classes



```
interface MyInterface {
    public void sayHello();
}

class SomeClass {

    void someMethod() {

        MyInterface a = new MyInterface() {
            public void sayHello() {
                System.out.println("Aye up");
            }
        };

        a.sayHello();

    }
}
```

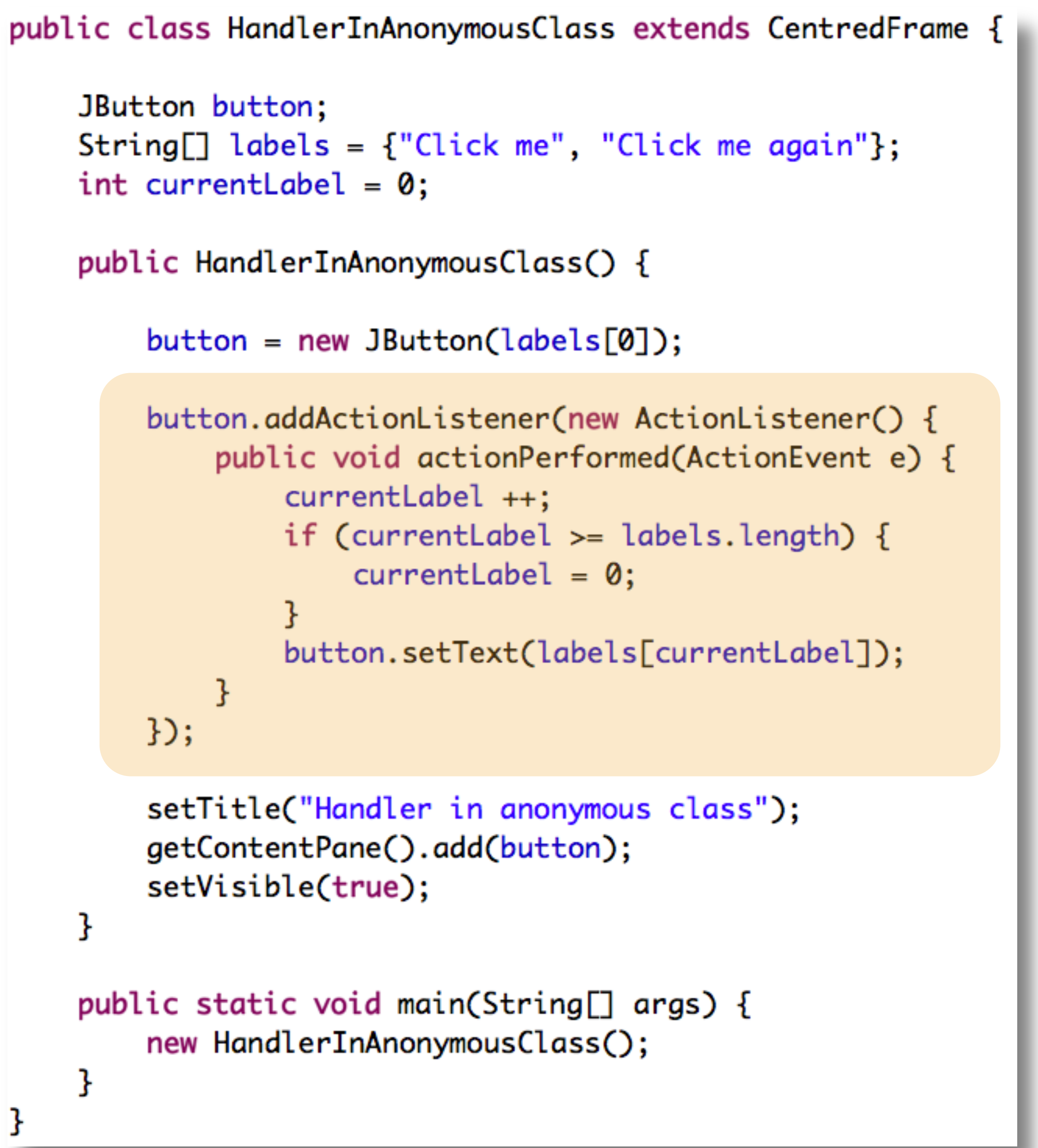
Anonymous classes implement or extend another class, but themselves have no name.

As they have no name, they cannot have a constructor.

Like local classes, they cannot have an access modifier.

Useful when the implementation of a class is very simple and only needs one instantiation.

Anonymous Classes



```
public class HandlerInAnonymousClass extends CentredFrame {

    JButton button;
    String[] labels = {"Click me", "Click me again"};
    int currentLabel = 0;

    public HandlerInAnonymousClass() {

        button = new JButton(labels[0]);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                currentLabel ++;
                if (currentLabel >= labels.length) {
                    currentLabel = 0;
                }
                button.setText(labels[currentLabel]);
            }
        });

        setTitle("Handler in anonymous class");
        getContentPane().add(button);
        setVisible(true);
    }

    public static void main(String[] args) {
        new HandlerInAnonymousClass();
    }
}
```

Anonymous classes implement or extend another class, but themselves have no name.

As they have no name, they cannot have a constructor.

Like local classes, they cannot have an access modifier.

Useful when the implementation of a class is very simple and only needs one instantiation.

A further extension of this idea is the use of lambda expressions – see the Java tutorials for details https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

COM1003 Java Programming
Richard Clayton

Back to GUIs and event handling

Steps for Creating and Wiring up a GUI Component

1. Instantiate the component
2. Add it to the container (else it won’t appear!)
3. Write an event listener
4. Register the listener with the component (easily forgotten!)

Further Types of Event

Low-level

ContainerEvent	Adding or removing a component
FocusEvent	Gain or lose focus
KeyEvent	Pressing or releasing a key
MouseEvent	Pressing, releasing, moving, dragging
WindowEvent	Opening, iconified, etc.

Semantic events

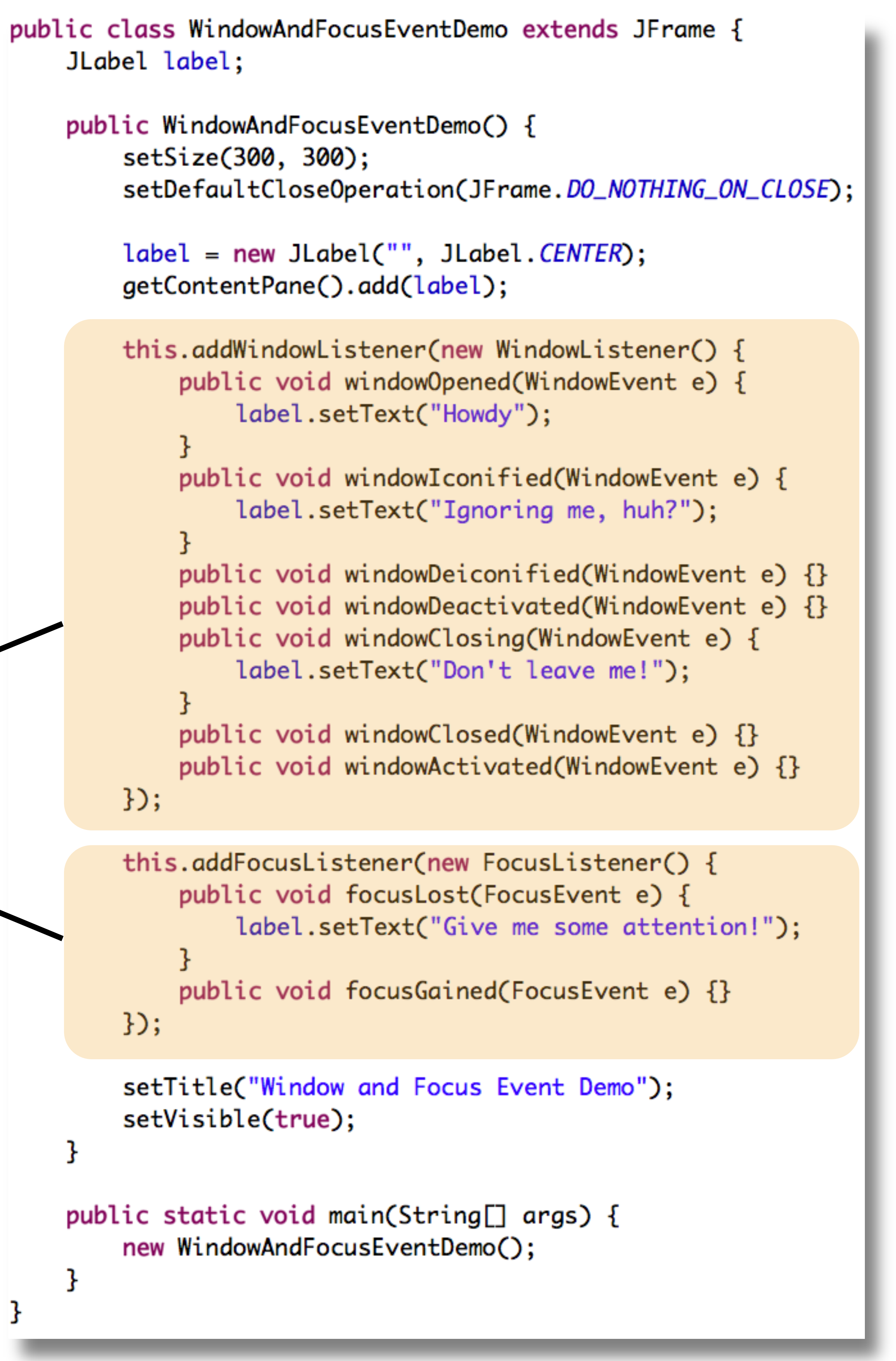
ActionEvent	Button click, menu selection, list item selection, hitting <ENTER>
AdjustmentEvent	Scroll bar adjusted
ItemEvent	Selection from set of checkbox items
TextEvent	Contents of text field changed

Listeners and Events

Interface	Methods	Event	Generated by
ActionListener	actionPerformed	ActionEvent	Component
ContainerListener	componentAdded componentRemoved	ContainerEvent	Container
FocusEvent	focusGained focusLost	FocusEvent	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
WindowListener	windowActivated windowOpened windowIconified windowDeiconified windowDeactivated windowClosing windowClosed	WindowEvent	Window

Handling WindowEvents & FocusEvents

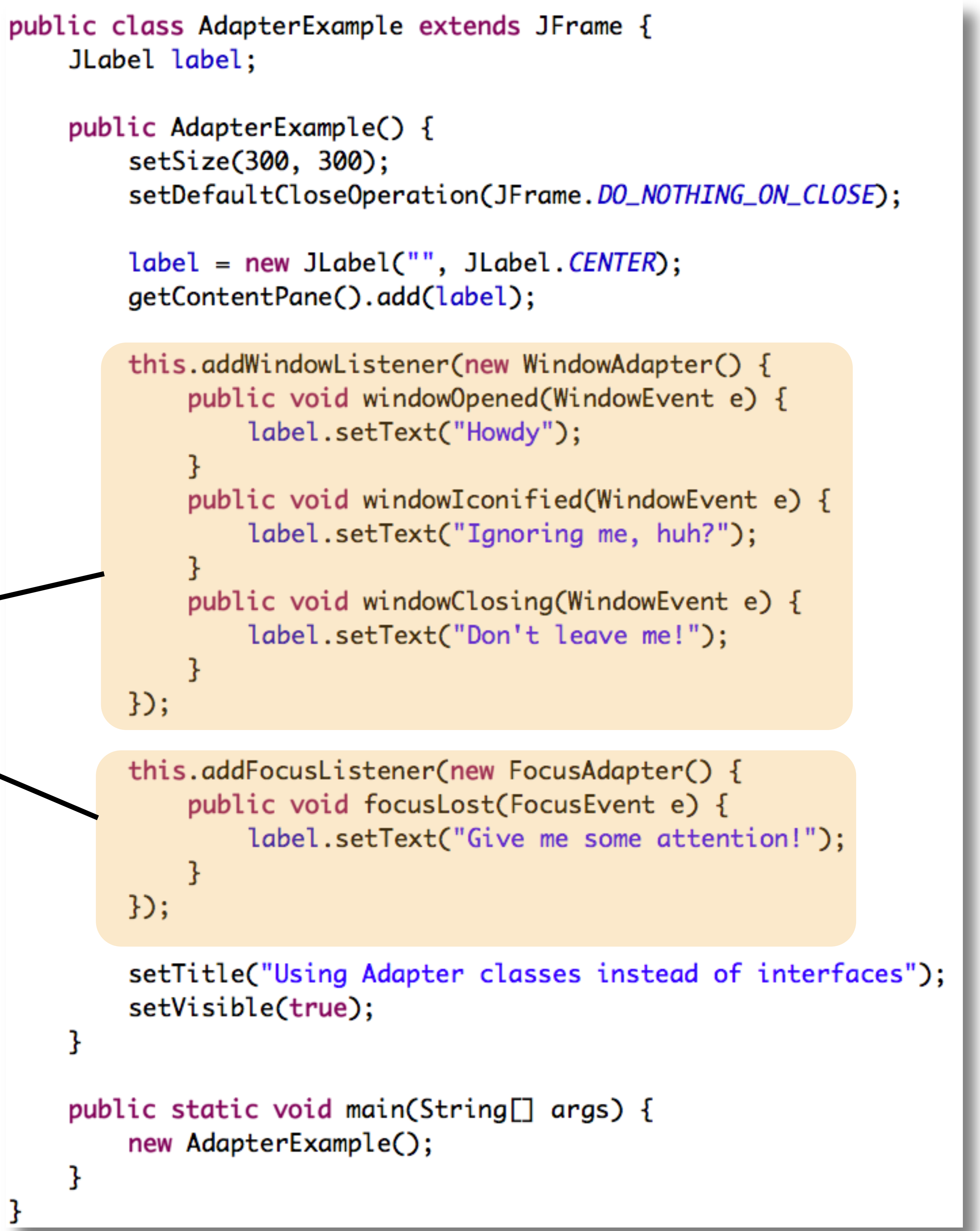
Notice that we did not need/want to override all the methods



```java
public class WindowAndFocusEventDemo extends JFrame {
    JLabel label;

    public WindowAndFocusEventDemo() {
        setSize(300, 300);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

        label = new JLabel("", JLabel.CENTER);
        getContentPane().add(label);

        this.addWindowListener(new WindowListener() {
            public void windowOpened(WindowEvent e) {
                label.setText("Howdy");
            }
            public void windowIconified(WindowEvent e) {
                label.setText("Ignoring me, huh?");
            }
            public void windowDeiconified(WindowEvent e) {}
            public void windowDeactivated(WindowEvent e) {}
            public void windowClosing(WindowEvent e) {
                label.setText("Don't leave me!");
            }
            public void windowClosed(WindowEvent e) {}
            public void windowActivated(WindowEvent e) {}
        });

        this.addFocusListener(new FocusListener() {
            public void focusLost(FocusEvent e) {
                label.setText("Give me some attention!");
            }
            public void focusGained(FocusEvent e) {}
        });

        setTitle("Window and Focus Event Demo");
        setVisible(true);
    }

    public static void main(String[] args) {
        new WindowAndFocusEventDemo();
    }
}
```

Adapter classes

Most listeners have a companion Adapter class, which is an abstract class implementing the interface with empty method bodies. So if we extend the adapter class instead, there's no need to add the empty method bodies ourselves.



```java
public class AdapterExample extends JFrame {
    JLabel label;

    public AdapterExample() {
        setSize(300, 300);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

        label = new JLabel("", JLabel.CENTER);
        getContentPane().add(label);

        this.addWindowListener(new WindowAdapter() {
            public void windowOpened(WindowEvent e) {
                label.setText("Howdy");
            }
            public void windowIconified(WindowEvent e) {
                label.setText("Ignoring me, huh?");
            }
            public void windowClosing(WindowEvent e) {
                label.setText("Don't leave me!");
            }
        });

        this.addFocusListener(new FocusAdapter() {
            public void focusLost(FocusEvent e) {
                label.setText("Give me some attention!");
            }
        });

        setTitle("Using Adapter classes instead of interfaces");
        setVisible(true);
    }

    public static void main(String[] args) {
        new AdapterExample();
    }
}
```

Events, Listeners and Adapters

Note the pattern:

A component generates an 'XEvent' which is handled by adding an implementation of the 'XListener' interface (or XAdapter class) using the method 'addXListener'.

In order to handle events for a given component, follow these rules:

1. Find out what kind of events the component generates
2. Choose the appropriate event
3. Look up the name of the corresponding listener interface class
4. Check if there is an adapter class which could be used instead
5. Check what kinds of parameters are passed to the listener methods
   (i.e. the details of the event itself)