Asynchronous Event Handling

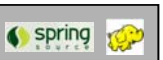
coreservlets.com – custom onsite training

Topics in This Section

- **General strategy for handling mouse clicks**
- **Four variations**
  - Handling events with separate listeners
  - Handling events by implementing interfaces
  - Handling events with named inner classes
  - Handling events with anonymous inner classes
- **Pros and cons of the options**
- **Preview: handling events with lambdas**
- **Some event-handler details**





Responding to Mouse Presses

Overview

- **You never explicitly check for events**
  - Instead, you simply register a mouse event handler
    `addMouseListener(yourMouseListener);`
  - Java automatically starts a separate thread to look for events, and when a mouse event occurs, that thread automatically calls the appropriate methods in your handler
- **Your handler must have all the MouseListener methods**
  - Even if you only care about mouse presses (mousePressed method), other methods will still be called. E.g., the mouseReleased method of your handler will still be called, even if you do not care about mouse releases.
  - Java confirms that you have all the necessary methods by checking at compile time that the argument to addMouseListener is a MouseListener

Alternative Implementation Strategies

- **addMouseListener(new SeparateClass(...));**
  - A separate class that implements MouseListener has mousePressed, mouseReleased, etc.
- **addMouseListener(this);**
  - The JPanel itself (or other window) has mousePressed, mouseReleased, etc.
  - The JPanel must then implement the MouseListener interface
- **addMouseListener(new InnerClass(...));**
  - An inner class that implements MouseListener has mousePressed, mouseReleased, etc.
- **addMouseListener(new MouseAdapter() {**
  // Body of anonymous inner class
  });
  - You define and instantiate the listener all at once

Alternatives for Mouse Listener Base

- **MouseListener**
  - Interface that defines 5 abstract methods: mouseEntered, mouseExited, mousePressed, mouseReleased, mouseClicked
    - Don't use mouseClicked; it means a release where the mouse did not move "much" since it was pressed. Use mousePressed or mouseReleased instead.
  - Your listener class must implement all 5 methods
- **MouseAdapter**
  - Abstract class that *already* implements the MouseListener interface and defines concrete versions of all 5 methods (with empty method bodies)
  - Your listener class can extend MouseAdapter, override the method you care about, and ignore the others

Techniques are Widely Applicable

- **Handler for window events**
  - Separate classes
  - Main class implementing interface
  - Named inner classes
  - Anonymous inner classes
  - Lambdas
- **Applications with same options and pros/cons**
  - Handler for GUI controls (buttons, sliders, etc.)
  - "Handler" (code to run in background) for multithreaded programming
  - "Handler" (code to compare elements) for array sorting
  - Handlers for code to be profiled, numeric integration, and lots more







Using Separate Listener Classes: Simple Case

Simple Case: No Access to Main Class

- **Goal**
  - Print out the location where mouse is pressed
- **Approach**
  - Define separate class as mouse listener
  - This class can extend MouseAdapter, override mousePressed, and ignore the other four methods
    - Inherits the other four methods, but the builtin version from MouseAdapter does nothing. I.e., all five methods are concrete (not abstract) in MouseAdapter, but have empty bodies.
  - The mousePressed method gets the x and y locations from the event passed to the method. It then just prints, so it needs no access to the JPanel instance.

Core Code

- **Panel**

```
public class ClickPanel extends JPanel {
  public ClickPanel() {
    setBackground(Color.YELLOW);
    addMouseListener(new ClickListener());
  }
}
```

- **Listener**

```
public class ClickListener extends MouseAdapter {
  @Override
  public void mousePressed(MouseEvent event) {
    int x = event.getX();
    int y = event.getY();
    System.out.printf("Mouse pressed at (%s, %s).%n", x, y);
  }
}
```

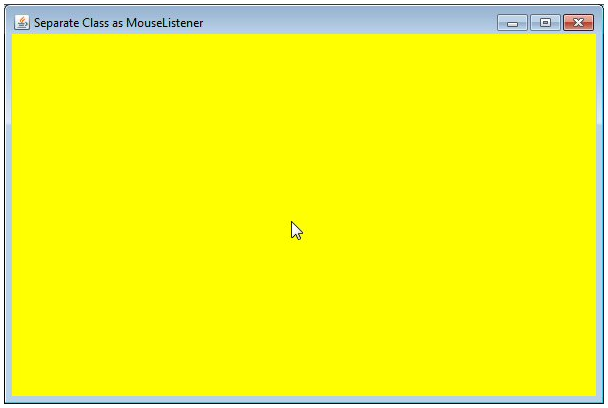

JFrame Code

```
public class ClickFrame extends JFrame {
  public ClickFrame() {
    super("Separate Class as MouseListener");
    setContentPane(new ClickPanel());
    setSize(600, 400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

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

Results

- **Representative output**

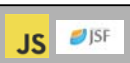
```
Mouse pressed at (10, 9).
Mouse pressed at (23, 24).
Mouse pressed at (49, 43).
Mouse pressed at (95, 85).
Mouse pressed at (147, 114).
Mouse pressed at (168, 139).
Mouse pressed at (218, 158).
Mouse pressed at (245, 185).
Mouse pressed at (290, 179).
```



Generalizing Simple Case

- **Question**
  - What if ClickListener wants to draw a circle wherever mouse is clicked?
  - Why can't it just manipulate some data structure used by the JPanel, then call the JPanel's repaint method?
- **Answer**
  - Because the ClickListener code has no variable referring to the JPanel
- **General solution**
  - Pass JPanel reference to the listener constructor
    - Within mousePressed, you can also use event.getSource() to get a reference to the window that the event came from
- **Deficiencies of this solution**
  - Methods called in the JPanel must be public
  - Code is a bit cumbersome







Shared Code for All Remaining Examples

Overview

- **Behavior of all examples**
  - A colored circle is drawn wherever user presses mouse
- **Code used by all variations**
  - Circle
    - A simple class that stores x, y, and radius. Also has a draw method that uses a supplied Graphics object to draw a circle of the appropriate size centered on (x,y).
  - CirclePanel
    - A class that extends JPanel and stores a List<Circle>. The paintComponent method loops down the list and calls draw on each circle. Each example will extend this class.
  - JFrame
    - Each of the examples will have an almost-identical JFrame that assigns a CirclePanel subclass as the content pane.

Circle

```
public class Circle {
  private final int x, y, radius;

  public Circle(int x, int y, int radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
  }

  // Getter methods: getX, getY, getRadius

  public void draw(Graphics g) {
    g.fillOval(x - radius, y - radius, radius * 2, radius * 2);
  }
}
```


CirclePanel: Core Methods

```
public class CirclePanel extends JPanel {
  protected int radius = 25;
  protected List<Circle> circles = new ArrayList<>();

  public CirclePanel() {
    setBackground(Color.YELLOW);
    setForeground(Color.RED);
  }

  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    for(Circle c: circles) {
      c.draw(g);
    }
  }
}
```

CirclePanel: Extra Methods

```
public int getRadius() {
  return(radius);
}

public List<Circle> getCircles() {
  return(circles);
}
```

- **Only needed to support separate listeners**
  - In first example, we will use a separate listener class. This class needs access to the circle radius and the circle List. We supply a reference to the main class in the listener's constructor, but the listener can only access public data.
    - So, these methods are needed *only* for that first example

CircleFrame (Representative Example)

```
public class CircleFrame1 extends JFrame {
  public CircleFrame1() {
    super("Separate Class as MouseListener");
    setContentPane(new CirclePanel1());
    setSize(600, 400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args) {
    new CircleFrame1();
  }
}
```

The JFrame code for each of the examples is almost identical, so will not be repeated. Complete source code is on the Web site, as always.






Using Separate Listener Classes

Panel

```
public class CirclePanel1 extends CirclePanel {
  public CirclePanel1() {
    radius = 10;
    addMouseListener(new CircleListener(this));
  }
}
```

Reminder: the parent CirclePanel class has public getRadius and getCircles methods.

Those methods are needed for this example only; the next three examples do not require public data.

Listener

```
public class CircleListener extends MouseAdapter {
  private CirclePanel window;

  public CircleListener(CirclePanel window) {
    this.window = window;
  }

  @Override
  public void mousePressed(MouseEvent event) {
    int x = event.getX();
    int y = event.getY();
    int radius = window.getRadius();
    window.getCircles().add(new Circle(x, y, radius));
    window.repaint();
  }
}
```

The listener stores a reference to the main window and uses that reference to call public methods.

Results

Reminder: Value of @Override

- **No @Override**

```
public void mousepressed(MouseEvent e) { ... }
```

  - No compile-time error
  - Nothing happens at run time when you press mouse

- **Using @Override**

```
@Override
public void mousepressed(MouseEvent e) { ... }
```

  - Compile-time error: no matching method mousepressed (because real method is mousePressed)

Implementing a Listener Interface

Review of Interfaces

- **Benefits**
  - Guarantees that classes will have certain methods
  - Objects can be treated as interface type
  - Classes can implement multiple interfaces
- **Example**

```
public interface Shape {
  double getArea(); // Method specification

  public static double sumAreas(Shape[] shapes) {
    double sum = 0;
    for(Shape s: shapes) {
      sum = sum + s.getArea();
    }
    return(sum);
  }
}
```


Source Code for MouseListener and MouseAdapter (Simplified)

```
public interface MouseListener {
  public void mouseClicked(MouseEvent e);
  public void mousePressed(MouseEvent e);
  public void mouseReleased(MouseEvent e);
  public void mouseEntered(MouseEvent e);
  public void mouseExited(MouseEvent e);
}
```

```
public abstract class MouseAdapter implements MouseListener {
  public void mouseClicked(MouseEvent e) {}
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}
}
```

Panel Part 1

```
public class CirclePanel2 extends CirclePanel
                          implements MouseListener {
  public CirclePanel2() {
    radius = 20;
    addMouseListener(this);
  }

  @Override
  public void mousePressed(MouseEvent event) {
    int x = event.getX();
    int y = event.getY();
    circles.add(new Circle(x, y, radius));
    repaint();
  }
```

Good news: since mousePressed is in the panel class, it can access protected (or even private) data members, and there is no need to pass along and store a reference.

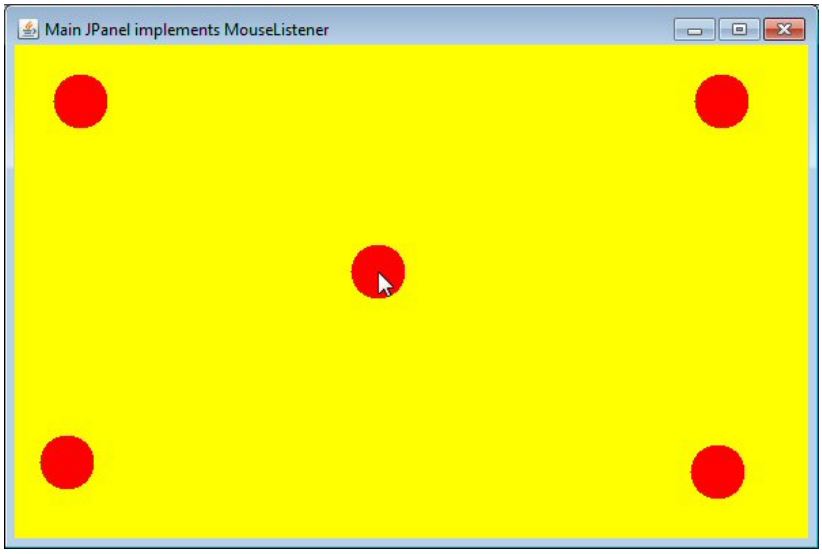
Panel Part 2

```
  @Override
  public void mouseClicked(MouseEvent event) {}

  @Override
  public void mouseReleased(MouseEvent event) {}

  @Override
  public void mouseEntered(MouseEvent event) {}

  @Override
  public void mouseExited(MouseEvent event) {}
}
```

Bad news: since you said you implement MouseListener, you must have all the MouseListener methods, even the ones you do not care about.

However, Eclipse can help. When you implement an interface, Eclipse can stub out the methods for you. R-click inside the class, Source, Override/Implement Methods.

Results



Using Named Inner Classes

Inner Classes: Overview

- **Class can be defined inside another class**
  - Methods in the inner class can access all methods and instance variables of surrounding class
    - Even private methods and variables
- **Advantages relative to the separate class approach**
  - No need to pass along a reference
  - You can access protected and private methods and variables of surrounding class
- **Advantages relative to the interface approach**
  - Inner class can extend another class (e.g., MouseAdapter)
  - Inner class can have constructors so that you can pass arguments to customize its behavior

Inner Classes: Quick Example

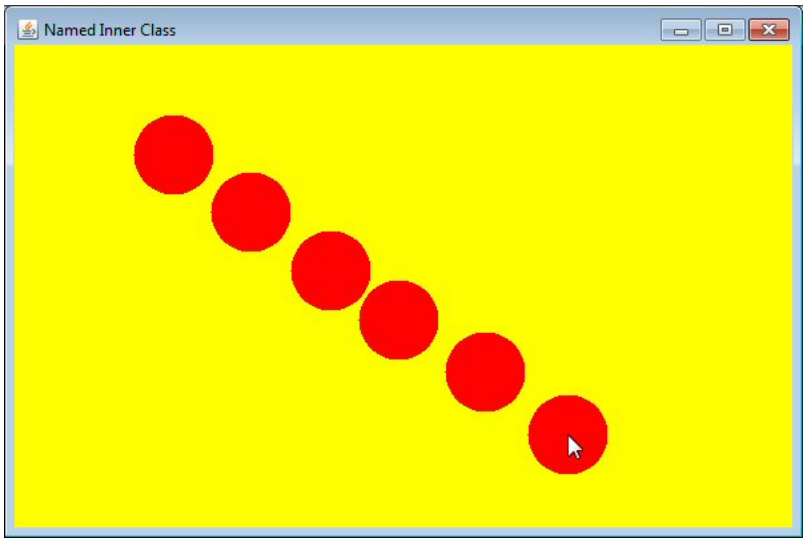
```
public class OuterClass {
  private int count = ...;

  public void foo(…) {
    InnerClass inner = new InnerClass();
    inner.bar();
  }

  private class InnerClass extends Blah {
    public void bar() {
      doSomethingWith(count);
    }
  }
}
```

Panel

```
public class CirclePanel3 extends CirclePanel {
  public CirclePanel3() {
    radius = 30;
    addMouseListener(new CircleListener());
  }

  private class CircleListener extends MouseAdapter {
    @Override
    public void mousePressed(MouseEvent event) {
      int x = event.getX();
      int y = event.getY();
      circles.add(new Circle(x, y, radius));
      repaint();
    }
  }
}
```

Good news: you can extend MouseAdapter for the listener. And, the mousePressed method can access protected (or private) data from the enclosing class.

Results



Using Anonymous Inner Classes

Anonymous Inner Classes: Overview

- **Class can be defined and instantiated all at once**
  - As with named inner classes, methods in inner class can access methods and instance variables of surrounding class
    - If methods of inner class access local variables of surrounding method, those variables cannot be modified (i.e., must be "effectively final")
- **Advantages vs. separate classes and interfaces**
  - Same as for named inner classes
- **Advantages vs. named inner classes**
  - Shorter
- **Disadvantages vs. named inner classes**
  - More confusing to beginners
  - No constructors
  - Cannot be reused elsewhere

Anonymous Inner Classes: Quick Example

```
public class OuterClass {
  private int count = …;

  public void foo(…) {
    SomeType inner = new SomeType() {
      public void bar() {
        doSomethingWith(count);
      }
    };
    inner.bar();
  }
}
```

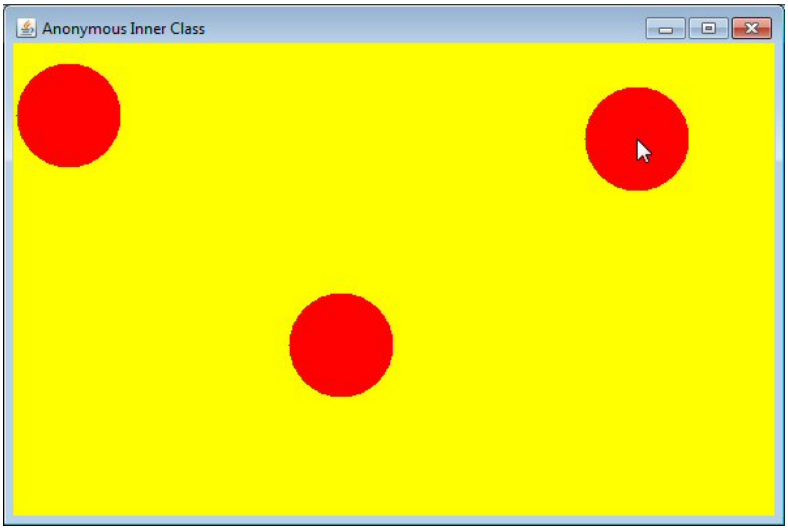
If SomeType is a class, read this as "I am instantiating a subclass of SomeType. I didn't give that subclass a name, but here is what it looks like".

If SomeType is an interface, read this as "I am instantiating a class that implements SomeType. I didn't give that class a name, but here is what it looks like".

Panel

```
public class CirclePanel4 extends CirclePanel {
  public CirclePanel4() {
    radius = 40;
    addMouseListener(new MouseAdapter() {
      @Override
      public void mousePressed(MouseEvent event) {
        int x = event.getX();
        int y = event.getY();
        circles.add(new Circle(x, y, radius));
        repaint();
      }
    });
  }
}
```

Results



Summary of Approaches

Event Handling Strategies: Pros and Cons

- **Separate Listener**
  - Advantages
    - Can extend adapter and thus ignore unused methods
    - Can pass arguments to class constructor
    - Separate class is more reusable
  - Disadvantages
    - Need extra step to call methods in main window, and those methods must be public
- **Main window that implements interface**
  - Advantages
    - Can easily call methods in main window, even private ones
  - Disadvantages
    - Must implement methods you might not care about
    - Hard to have multiple different versions since you cannot pass arguments to listener

Event Handling Strategies: Pros and Cons (Continued)

- **Named inner class**
  - Advantages
    - Can extend adapter and thus ignore unused methods
    - Can easily call methods in main app, even private ones
    - Can define constructor and pass arguments
  - Disadvantage
    - A bit harder to understand
- **Anonymous inner class**
  - Advantages
    - Same as named inner classes, but shorter
  - Disadvantage
    - Harder to understand for beginners
    - Not reusable
    - No constructors

Preview: Using Java 8 Lambdas

Overview

- **Features**
  - Full access to code from surrounding class
  - No confusion about meaning of "this"
  - Much more concise, succinct, and readable
  - Encourage a functional programming style
  - But no instance variables, so lambdas are not *always* better
- **You can*not* use lambdas for mouse listeners**
  - Because MouseListener has multiple methods, and lambdas can be used only for 1-method interfaces
  - But, you could use lambdas for ActionListener, which applies to push buttons (see next slide)

Quick Example (Details in Upcoming Sections)

- **Anonymous inner class**

```
myButton.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent event) {
    doSomethingWith(event);
  }
});
```

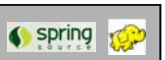
- **Lambda**

```
myButton.addActionListener(event -> doSomethingWith(event));
```









Event Handling Details

Mouse Event Details

- **Which mouse button**
  - SwingUtilities.isLeftMouseButton(event)
  - SwingUtilities.isRightMouseButton(event)
- **Modifier keys**
  - event.isControlDown, event.isShiftDown, etc.
    - Macs with a one-button mouse often use control-click as equivalent of right click. So, you can detect right click equivalents portably like this:

      if (SwingUtilities.isRightMouseButton(event) || event.isControlDown()) { ... }
- **Single vs. double clicks**
  - event.getClickCount
    - For a double click, it fires mousePressed twice, once with a click count of 1 and then again with a click count of 2

Other Low-Level Events

- **Moving/dragging mouse**
  - Use MouseMotionListener or MouseMotionAdapter
    - mouseMoved: mouse moved while button was up
    - mouseDragged: mouse moved while button was down
- **Typing on the keyboard**
  - Use KeyListener or KeyAdapter
    - keyTyped: key was released for a printable character
    - Get String with String.valueOf(event.getKeyChar());
    - JPanel normally ignores keyboard events. If you want it to get them, you must do this:

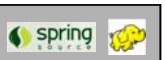
```
setFocusable(true);
requestFocusInWindow();  // Or click on window
                         // before typing
```

Higher-Level Events

- **Most GUI controls have higher-level events**
  - Buttons use ActionListener
    - Button was triggered by any of clicking it, hitting Enter while it has focus, or keyboard shortcut
  - Checkboxes, radio buttons, and others use ItemListener
    - To respond when the state has changed
  - Text controls automatically handle keyboard events
    - But you can attach TextListener to respond to changes
  - Frames and dialog boxes use WindowListener
    - To detect when user attempts to close window. E.g., to pop up confirmation like "Unsaved changes; really quit?"
  - JEditorPane (to render HTML) has HyperLinkListener
    - To respond when user clicks on a link





Wrap-Up

Summary

- **Register a mouse event handler**

```
addMouseListener(yourMouseListener);
```

- **Four variations**

```
addMouseListener(new SeparateClass(...));
```

– Reusable, but hard to call back to main class

```
addMouseListener(this);
```

– Easy to access main class, but no constructors

```
addMouseListener(new InnerClass(...));
```

– Easy to access main class, also can use constructors

```
addMouseListener(new MouseAdapter() { ... });
```

– Easy to access main class, shorter, but not reusable