Events (in GUI Programming)

- All GUI applications are event-driven.
- In GUI programming, events describe the change in the state of a GUI component when users interact with it
  - A button is clicked
  - The mouse is moved
  - A character is entered through keyboard
  - An item from a list is selected
  - ……

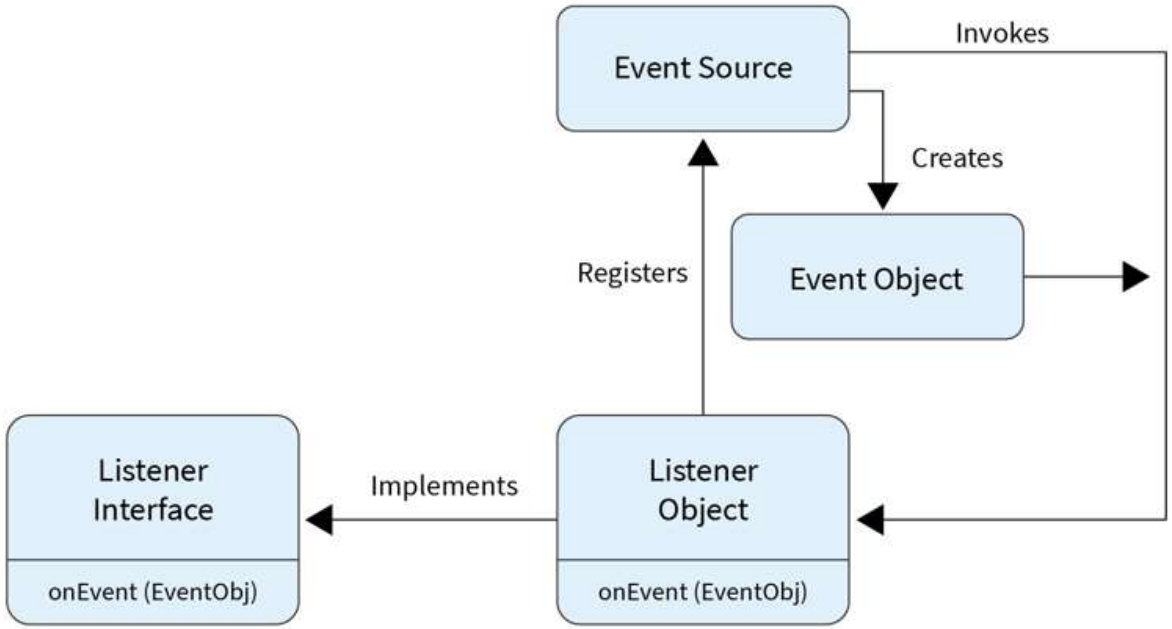
Event Handling

- Event handling is the mechanism that controls the event and decides what should happen if an event occurs. Three key concepts:
  - **Event source (事件源):** the GUI component with which the user interacts (e.g., a button)
  - **Event object (or simply event):** encapsulate the information about the event that occurred (e.g., a MouseEvent)
  - **Event listener (事件监听器):** an object that is notified by the event source when an event occurs.
    - A method of the event listener receives an event object when the event listener is notified of the event.
    - The listener then uses the event object to respond to the event.

Delegation Event Model (事件委托模型)

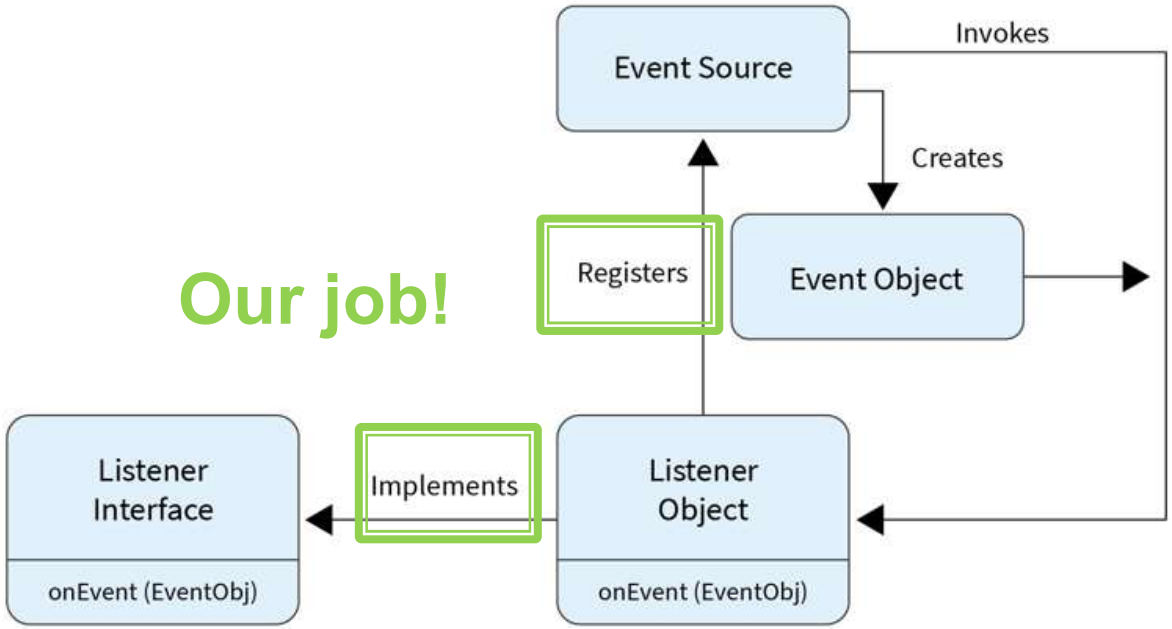
UI components delegate an event's processing to an event listener object

- A source can register one or more listeners to receive notifications for specific events.
- A source generates an event and forwards it to one or more listeners.
- The listener waits until it receives an event and react properly using the info in the event object



Delegation Event Model (事件委托模型)

- As developers, we need to:
  - Create the listener by implementing the Listener interface
  - Register the listener to the event source
- We don't need to worry about how event object is created and how listeners are invoked



Event Classes and Listener Interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Event Handling Example

- We use a counter program to illustrate the steps

```
public class SwingCounter extends JFrame {
    private JTextField tfCount;
    private JButton btnCount;
    private int count = 0;
    public SwingCounter() {
        setLayout(new FlowLayout(FlowLayout.LEFT, 50, 0));
        add(new JLabel("Counter"));
        tfCount = new JTextField("0");
        tfCount.setEditable(false); add(tfCount);
        btnCount = new JButton("Count"); add(btnCount);
    }
    public static void main(String[] args) { SwingCounter sc = new SwingCounter(); ... }
}
```

Nothing will happen when we click the button (we have not handled the event yet)

Event Handling Example

- **Step 1:** check what event will occur when JButton is clicked

- An `ActionEvent` (in `java.awt.event` package) will occur whenever the user performs a component-specific action on a GUI component
  - When user clicks a button
  - When user chooses a menu item
  - When user presses Enter after typing something in a text field…

Event Handling Example

- **Step 2:** define the event listener class by implementing the corresponding listener interface

```
public class ButtonClickListener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent arg0) {
        // code to react to the event
    }

}
```

`ActionListener` is from the package `java.awt.event`

Event Handling Example

- The event listener class is often declared as an inner class

```
public class SwingCounter extends JFrame {

    private JTextField tfCount;
    private JButton btnCount;
    private int count = 0;

    public class ButtonClickListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent arg0) {
            ++count;  tfCount.setText(count + "");
        }

    }
}
```

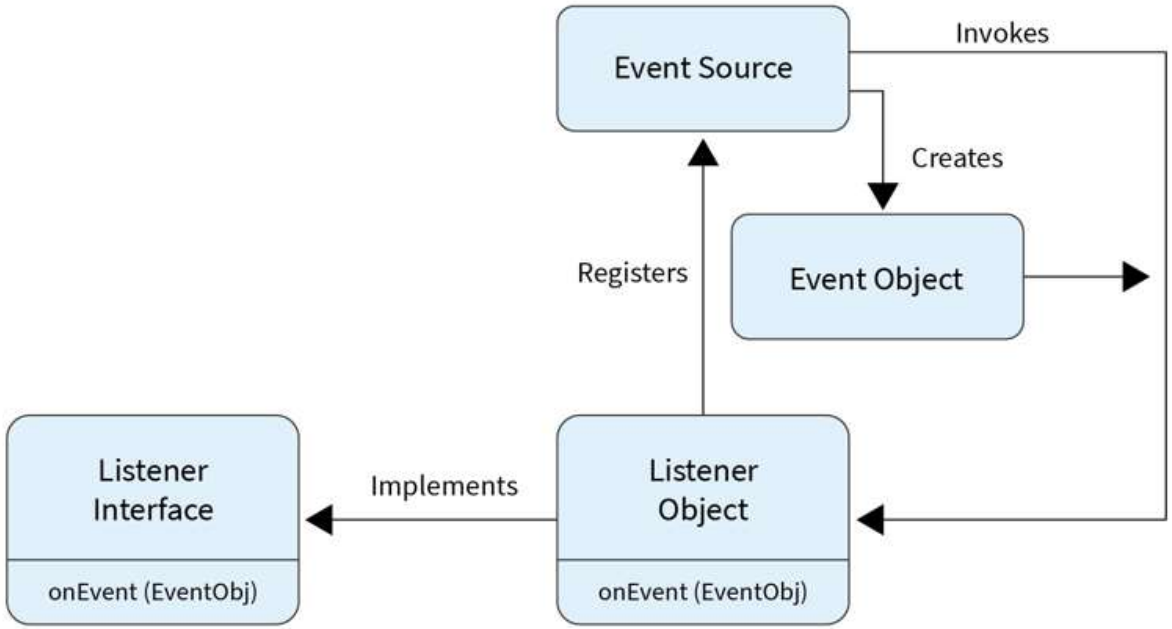
An inner class is a proper class. It can have constructors, fields, methods …

An inner class is a member of the outer class. Therefore, it can access the private members of the outer class **(this is very useful)**

Event Handling Example

- **Step 3:** register an instance of the event listener class as a listener on the corresponding GUI component (event source)

```
btnCount.addActionListener(new ButtonClickListener());
```



```
public class SwingCounter extends JFrame {
    private JTextField tfCount;
    private JButton btnCount;  ← Event source
    private int count = 0;
    public SwingCounter() {
        setLayout(new FlowLayout(FlowLayout.LEFT, 50, 0));
        add(new JLabel("Counter"));
        tfCount = new JTextField("0");
        tfCount.setEditable(false); add(tfCount);
        btnCount = new JButton("Count"); add(btnCount);
        btnCount.addActionListener(new ButtonClickListener());  ← Event listener
    }
    public class ButtonClickListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent arg0) {
            count++; tfCount.setText(count + "");
        }
    }
    public static void main(String[] args) { … }
}
```

Event object will be passed here

Initial state

After one click

After two clicks

...

After 10 clicks

After 11 clicks

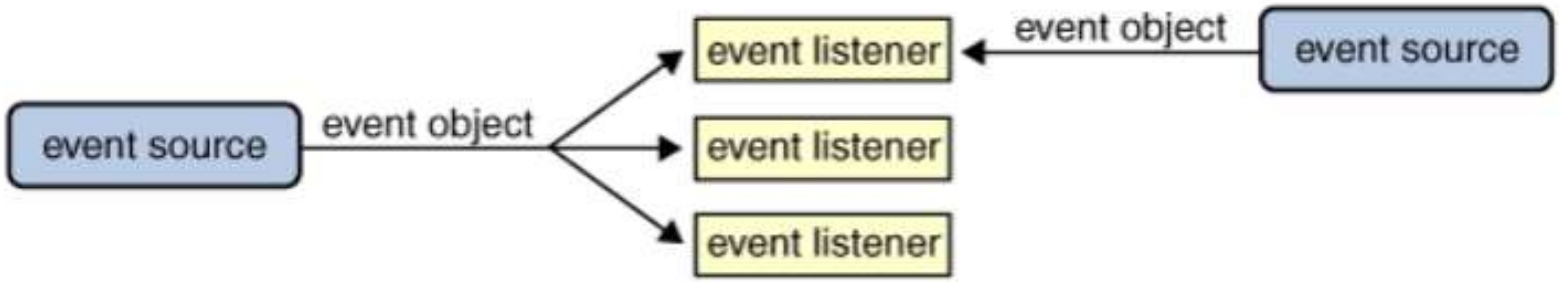
After 12 clicks

…

What's the problem?

Event Listeners

- A program can have one or more listeners for a single kind of event from a single event source.
- A program might have a single listener for all events from all sources (e.g., the calculator buttons).



https://docs.oracle.com/javase/tutorial/uiswing/events/intro.html

Implementing Event Listeners

- Inner class
  - A class defined within another class (outer class)
  - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
  - An inner class can access private members of the outer class
- Anonymous class
- Lambda expression

Implementing Event Listeners

- Anonymous class
  - Anonymous classes are inner classes with no name
  - We need to declare and instantiate anonymous classes in a single expression at the point of use.

```
new InterfaceName() {...}
```

name of the interface to implement — methods' implementations

```
btnCount.addActionListener(new ButtonClickListener());
```

```
public class ButtonClickListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        ++count;
        tfCount.setText(count + "");
    }
}
```

⇨

```
btnCount.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        ++count;
        tfCount.setText(count + "");
    }
});
```

Implementing Event Listeners

- Lambda Expression
  - To implement interfaces that have just one method, we could use lambda expressions

```
public interface ActionListener extends EventListener {

    public void actionPerformed(ActionEvent e);

}
```

```
btnCount.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        ++count;
        tfCount.setText(count + "");
    }
});
```

```
btnCount.addActionListener(e -> {
    ++count;
    tfCount.setText(count + "");
});
```

Simplifying code with lambda expressions

```java
public class SwingCounter extends JFrame {
   private JTextField tfCount;
   private JButton btnCount;
   private int count = 0;

   public SwingCounter() {
      setLayout(new FlowLayout(FlowLayout.LEFT, 50, 0));
      add(new JLabel("Counter"));
      tfCount = new JTextField("0");
      tfCount.setEditable(false);
      add(tfCount);
      btnCount = new JButton("Count");

      btnCount.addActionListener(new ButtonClickListener());

      add(btnCount);

   }

   public class ButtonClickListener implements ActionListener {
      @Override
      public void actionPerformed(ActionEvent arg0) {
         ++count;
         tfCount.setText(count + "");
      }

   }
```

```java
public class SwingCounter extends JFrame {
   private JTextField tfCount;
   private JButton btnCount;
   private int count = 0;

   public SwingCounter() {
      setLayout(new FlowLayout(FlowLayout.LEFT, 50, 0));
      add(new JLabel("Counter"));
      tfCount = new JTextField("0");
      tfCount.setEditable(false);
      add(tfCount);
      btnCount = new JButton("Count");

      btnCount.addActionListener(e -> {
         ++count;
         tfCount.setText(count + "");
      });

      add(btnCount);

   }
```

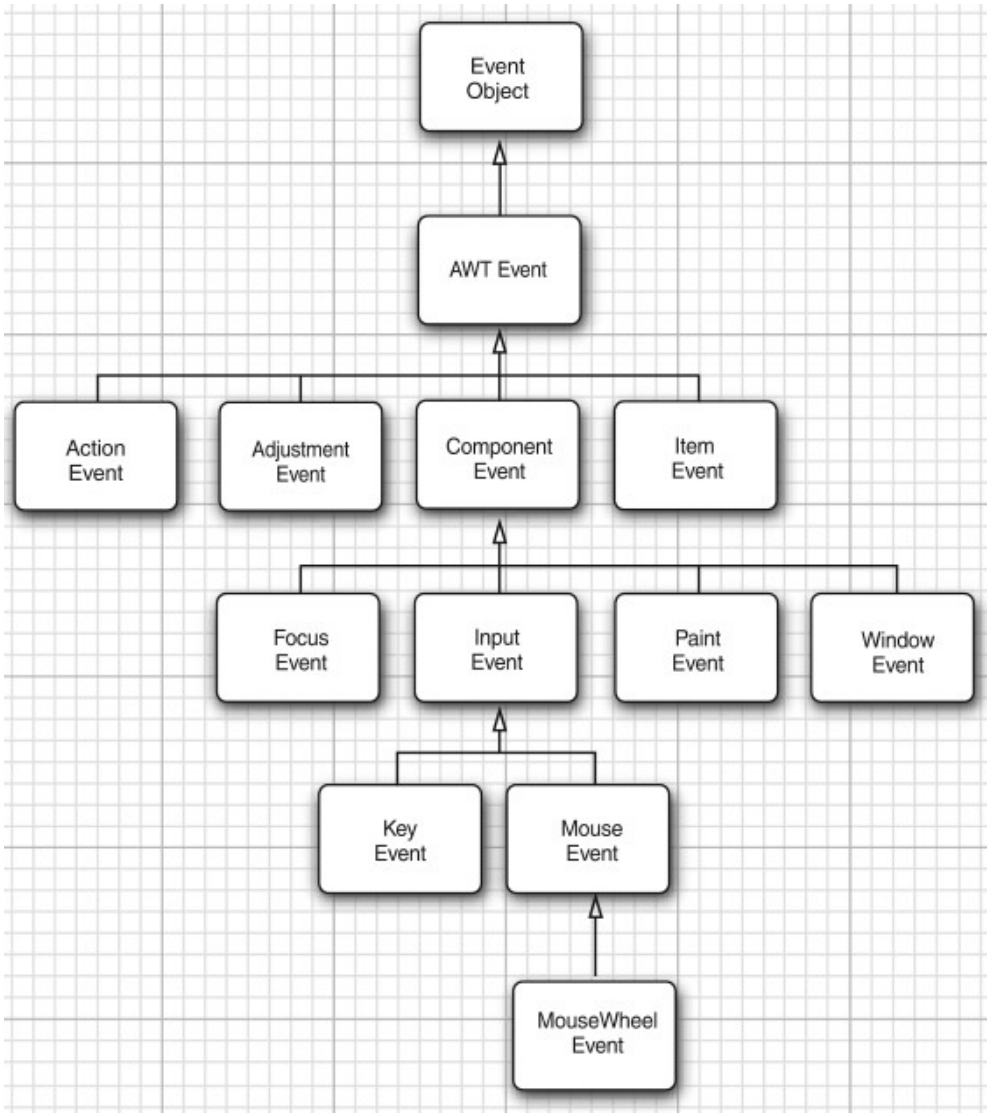
In Java, you can use Lambda expressions to simplify classes that implement interfaces that have just one method

The AWT Event Hierarchy

- The event objects encapsulate information about the event that the event source communicates to its listeners.

- When necessary, you can then analyze the event objects that were passed to the listener object

Reference: Core Java, Volume I. Chapter 10.4



Semantic & Low-level Events

- Semantic events: expresses what the user is doing
  - `ActionEvent`: e.g., button click, menu selection
  - `AdjustmentEvent`: e.g., adjust a scrollbar
  - `ItemEvent`: e.g., selecting from a list item or checkbox

- Low-level events: events that make semantic events possible
  - `KeyEvent`: e.g., a key is pressed or released
  - `MouseEvent`: e.g., a mouse is pressed, moved, or dragged
  - `MouseWheelEvent`
  - `FocusEvent`
  - `WindowEvent`

Reference: Core Java, Volume I. Chapter 10.4

Semantic & Low-level Events

Interface	Methods	Parameter/Accessors	Events Generated By
ActionListener	actionPerformed	ActionEvent • getActionCommand • getModifiers	AbstractButton JComboBox JTextField Timer
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent • getAdjustable • getAdjustmentType • getValue	JScrollbar
ItemListener	itemStateChanged	ItemEvent • getItem • getItemSelectable • getStateChange	AbstractButton JComboBox

Reference: Core Java, Volume I. Chapter 10.4

Semantic & Low-level Events

FocusListener	focusGained focusLost	FocusEvent • isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent • getKeyChar • getKeyCode • getKeyModifiersText • getKeyText • isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent • getClickCount • getX • getY • getPoint • translatePoint	Component

Reference: Core Java, Volume I. Chapter 10.4

Semantic & Low-level Events

Interface	Methods	Parameter/Accessors	Events Generated By
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent • getWheelRotation • getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent • getWindow	Window
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent • getOppositeWindow	Window

Should we implement all these methods in this interface even if we're interested in only one of them?

Reference: Core Java, Volume I. Chapter 10.4

Adapter Class

- Each AWT listener interface that has more than one method comes with a companion adapter class, which implements all methods in the interface but does nothing with them

- For example, `WindowAdapter` is an abstract adapter class for receiving window events. The methods in this class are empty. This class exists as convenience for creating listener objects.

Reference: Core Java, Volume I. Chapter 10.4

Adapter Class

- Extend this class to create a `WindowEvent` listener and override the methods for the events of interest.

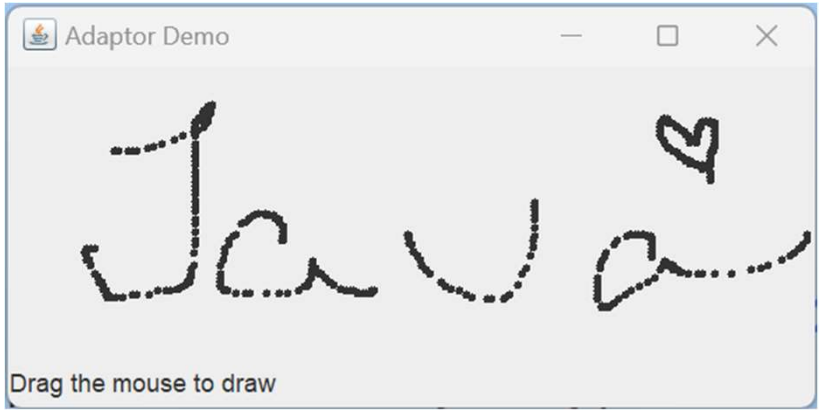
- If you implement the `WindowListener` interface, you have to define all of the methods in it. This abstract class defines null methods for them all, so you can only have to define methods for events you care about.

```
class Terminator extends WindowAdapter
{
   public void windowClosing(WindowEvent e)
   {
      if (user agrees)
         System.exit(0);
   }
}

WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

Reference: Core Java, Volume I. Chapter 10.4

Adaptor Example




```
public class AdaptorDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Adaptor Demo");

        PaintPanel paintPanel = new PaintPanel();
        frame.add(paintPanel, BorderLayout.CENTER);

        frame.add(new Label("Drag the mouse to draw"),
                                        BorderLayout.SOUTH);

        frame.setSize(400,200);
        frame.setVisible(true);
    }
}
```

Adaptor Example

- Class `PaintPanel` extends `JPanel` to create the dedicated drawing area.
- We use an `ArrayList` of `Point (java.awt)` to store the location at which each mouse drag event occurs

```
class PaintPanel extends JPanel{
    private ArrayList<Point> points = new ArrayList<>();

    PaintPanel(){
        addMouseMotionListener(new MouseMotionAdapter() {
            @Override
            public void mouseDragged(MouseEvent e) {
                points.add(e.getPoint());
                repaint();
            }
        });
    }

    @Override
    public void paintComponent(Graphics g){

        super.paintComponent(g);

        for(Point point: points){
            g.fillOval(point.x, point.y,4,4);
        }
    }
}
```

Adaptor Example

Register a `MouseMotionListener` to listen for the `PaintPanel`'s mouse motion events.

Override method `mouseDragged`: invoke the `MouseEvent`'s `getPoint()` to obtain the Point where the event occurred and stores it in the ArrayList.

```
class PaintPanel extends JPanel{
    private ArrayList<Point> points = new ArrayList<>();

    PaintPanel(){
        addMouseMotionListener(new MouseMotionAdapter() {
            @Override
            public void mouseDragged(MouseEvent e) {
                points.add(e.getPoint());
                repaint();
            }
        });
    }

    @Override
    public void paintComponent(Graphics g){

        super.paintComponent(g);

        for(Point point: points){
            g.fillOval(point.x, point.y,4,4);
        }
    }
}
```

Create an object of an anonymous inner class that extends the adapter class `MouseMotionAdapter` which implements `MouseMotionListener`

Adaptor Example

Calls `repaint()` (inherited indirectly from class `Component`) to indicate that the `PaintPanel` should be refreshed on the screen as soon as possible with a call to the `PaintPanel's` `paintComponent` method.

Invoke the superclass version of `paintComponent` to clear the `PaintPanel's` background

Draw an solid oval at the location specified by each Point in the ArrayList.

```
class PaintPanel extends JPanel{
    private ArrayList<Point> points = new ArrayList<>();

    PaintPanel(){
        addMouseMotionListener(new MouseMotionAdapter() {
            @Override
            public void mouseDragged(MouseEvent e) {
                points.add(e.getPoint());
                repaint();
            }
        });
    }

    @Override
    public void paintComponent(Graphics g){

        super.paintComponent(g);

        for(Point point: points){
            g.fillOval(point.x, point.y,4,4);
        }
    }
}
```

Read the Doc!

https://docs.oracle.com/javase/tutorial/uiswing/TOC.html

Trail: Creating a GUI With Swing: Table of Contents