Mouse Events

Slides adapted from slides of Marty Stepp based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

Mouse events

- Usage of mouse events:
  - listen to clicks / movement of mouse in a component
  - respond to mouse activity with appropriate actions
  - create interactive programs that are driven by mouse activity

- Note that the listener is assigned to a component not to the mouse itself. It tracks the mouse only while it is inside the component.

Mouse Events interfaces

- Often one is interested in the mouse buttons or in the mouse motion but not both.
- There are two listeners defined by the interfaces `MouseListener` and `MouseMotionListener` in the AWT library `java.awt.event`.

- `MouseListener` is for handling the mouse buttons while `MouseMotionListener` monitors the motion.
- To add a mouse listener to a Swing component “comp” use
  - `comp.addMouseListener(MouseListener mouseListener);`
  - `comp.addMouseMotionListener(MouseMotionListener motionListener);`

MouseListener interface

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

- Many AWT/Swing components have this method:
  - `public void addMouseListener(MouseListener ml)`
- Mouse listeners are often added to custom components and drawing canvases to respond to clicks and other mouse actions.

MouseListener Interface

- `mouseClicked(MouseEvent mevt)` is automatically called by the runtime system when a mouse button is clicked. To find out which button was clicked one has to look at the MouseEvent object “mevt”.
- `mouseEntered(MouseEvent mevt)` is automatically called by the runtime system if the mouse enters the component to which the listener is associated.
- `mouseExited(MouseEvent mevt)` is automatically called by the runtime system if the mouse leaves the component to which the listener is associated.
- `mousePressed(MouseEvent mevt)` is automatically called by the runtime system if a mouse button is pressed. To find out which button was pressed one has to look at the MouseEvent object “mevt”.
- `mouseReleased(MouseEvent mevt)` is automatically called by the runtime system if a mouse button is released. To find out which button was released one has to look at the MouseEvent object “mevt”.
- A click results in calling three methods, `mousePressed`, `mouseReleased` and `mouseClicked`.

- Interface `MouseListener` has five such methods, all of which have to be implemented

```
// elsewhere,
myComponent.addMouseListener(MouseListener mouseListener);
```

  - Problem: Tedious to implement entire interface when only partial behavior is wanted / needed.

MouseMotionListener Interface

- **`void`** `mouseMoved(MouseEvent mevt)`
- **`void`** `mouseDragged(MouseEvent mevt)`
- `mouseMoved(MouseEvent mevt)` is automatically called if the mouse is moved while it is in a component that the listener is associated to.
  - The new mouse position can be found by analyzing the event object “mevt”.
  - The mouse is ‘moved’ if the runtime system receives a signal from the mouse.
    - Usually this results in a change of the position of the mouse pointer on the screen by at least one pixel.
- `mouseDragged(MouseEvent mevt)` is automatically called if the mouse is moved while a mouse button is pressed.

- Interface `MouseMotionListener` has two methods, all of which have to be implemented

```
// elsewhere,
myComponent.addMouseMotionListener(MouseMotionListener motionListener);
```

Mouse Event

- The methods described all receive a (`MouseEvent`) as a parameter. This object contains information on what triggered the listener.
- The following methods from the class `MouseEvent` show where the event occurred. The *x*- and *y* coordinates are in pixels in the coordinate system of the component to which the listener is assigned.
  - **`int`** `getX()`
  - **`int`** `getY()`
  - `public Point getPoint()`

- The following methods from the class `SwingUtilities` are used to find out which button was used.
  - These methods return `true` if the button appearing in the method name is used, and `false` otherwise
  - **`boolean`** `SwingUtilities.isLeftMouseButton(MouseEvent me)`
  - **`boolean`** `SwingUtilities.isMiddleMouseButton(MouseEvent me)`
  - **`boolean`** `SwingUtilities.isRightMouseButton(MouseEvent me)`
- `public static int BUTTON1_MASK, BUTTON2_MASK, BUTTON3_MASK, CTRL_MASK, ALT_MASK,SHIFT_MASK`
- `public Object getSource()`
- `public int getModifiers() // use *_MASK with this`

- `int getClickCount()`
  - This method is used to determine the number of clicks in a short period.
  - The length of this period depends on the platform and can usually be set using utilities of the operating system.
  - Then a double click can be checked by using if(mevt.getClickCount() == 2)

Adapter pattern

- *Problem:* We have an object that contains the functionality we need, but not in the way we want to use it.
  - Cumbersome / unpleasant to use. Prone to bugs.

- *Example:*
  - We want to write one or two mouse input methods.
  - Java makes us implement an interface full of methods we don't want.

- *Solution:*
  - Provide an adapter class that connects into the setup we must talk to (GUI components) but exposes to us the interface we prefer (only have to write one or two methods).

Event adapters

- **event adapter**: A class with empty implementations of all of a given listener interface's methods.
  - examples: `MouseAdapter`, `KeyAdapter`, `FocusAdapter`
  - Extend `MouseAdapter`; override methods you want to implement.
    - Don't have to type in empty methods for the ones you don't want!

An abstract event adapter

```
// This class exists in package java.awt.event.

// An empty implementation of all MouseListener methods.
public abstract class MouseAdapter implements MouseListener,
  MouseMotionListener, MouseWheelListener {
    public void mousePressed(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
    public void mouseWheelMoved(MouseWheelEvent event){}
    public void mouseMoved(MouseEvent mevt) {}
    public void mouseDragged(MouseEvent mevt) {}

}
```

- Now classes can extend `MouseAdapter` rather than implementing `MouseListener`, `MouseMotionListener`, and `MouseWheelListener`.
  - gets the complete mouse listener interface it wants
  - implementer gets to write just the few mouse methods they want

Abstract classes

- **abstract class**: A hybrid between an interface and a class.
  - Defines a superclass type that can contain method declarations (like an interface) and/or method bodies (like a class).
  - Like interfaces, abstract classes that cannot be instantiated (cannot use `new` to create any objects of their type).

- What goes in an abstract class?
  - Implementation of common state and behavior that will be inherited by subclasses (parent class role)
  - Declare generic behavior for subclasses to implement (interface role)

Abstract class syntax

```
// declaring an abstract class
public abstract class name {
    ...

    // declaring an abstract method
    // (any subclass must implement it)
    public abstract type name(parameters);

}
```

- A class can be `abstract` even if it has no abstract methods
- You can create variables (but not objects) of the abstract type

Writing an adapter

```
public class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent event) {
        System.out.println("You pressed the button!");
    }
}

// elsewhere,
myComponent.addMouseListener(new MyMouseAdapter());
```

Abstract class vs. interface

- Why do both interfaces and abstract classes exist in Java?
  - An abstract class can do everything an interface can do and more.
  - So why would someone ever use an interface?

- Answer: Java has only single inheritance.
  - can extend only one superclass
  - can implement many interfaces

  - Having interfaces allows a class to be part of a hierarchy (polymorphism) without using up its inheritance relationship.

Using MouseEvent

```
public class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent event) {
        Object source = event.getSource();
        if (source == button && event.getX() < 10) {
            JOptionPane.showMessageDialog(null,
                "You clicked the left edge!");
        }
    }
}
```

Mouse motion

- The abstract `MouseMotionAdapter` class provides empty implementations of both methods if you just want to override one.

Mouse wheel scrolling

```
public interface MouseWheelListener {
    public void mouseWheelMoved(MouseWheelEvent event);
}
```

- The third mouse event interface focuses on the rolling of the mouse's scroll wheel button.

- Many AWT/Swing components have this method:
  - `public void addMouseWheelListener(MouseWheelListener ml)`

Mouse input listener

```
// import javax.swing.event.*;
public interface MouseInputListener
    extends MouseListener, MouseMotionListener
        {}
```

- The `MouseInputAdapter` class includes empty implementations for all methods from all mouse input interfaces, allowing the same listener object to listen to mouse clicks, movement, and/or wheel events.
  - `Extends MouseAdapter`
  - `Implements MouseInputListener`

Mouse input example

```
public class MyMouseInputAdapter extends MouseInputAdapter {
    public void mousePressed(MouseEvent event) {
        System.out.println("Mouse was pressed");
    }

    public void mouseDragged(MouseEvent event) {
        Point p = event.getPoint();
        System.out.println("Mouse is at " + p);
    }
}

...

// using the listener
MyMouseInputAdapter adapter = new MyMouseInputAdapter();
myPanel.addMouseListener(adapter);
myPanel.addMouseMotionListener(adapter);
```