CS416

Introduction to Computer Science II

Spring 2018

4 Simple Animation

- Previously, in 416
  - *JComponent*
  - *JWheels*
    - application level graphics objects that are *JComponents*

Preview

- Java event model
- Basic event-based programming in Java
- Simple time-based animation
  - Mover interface
  - Swing Timer class

Events

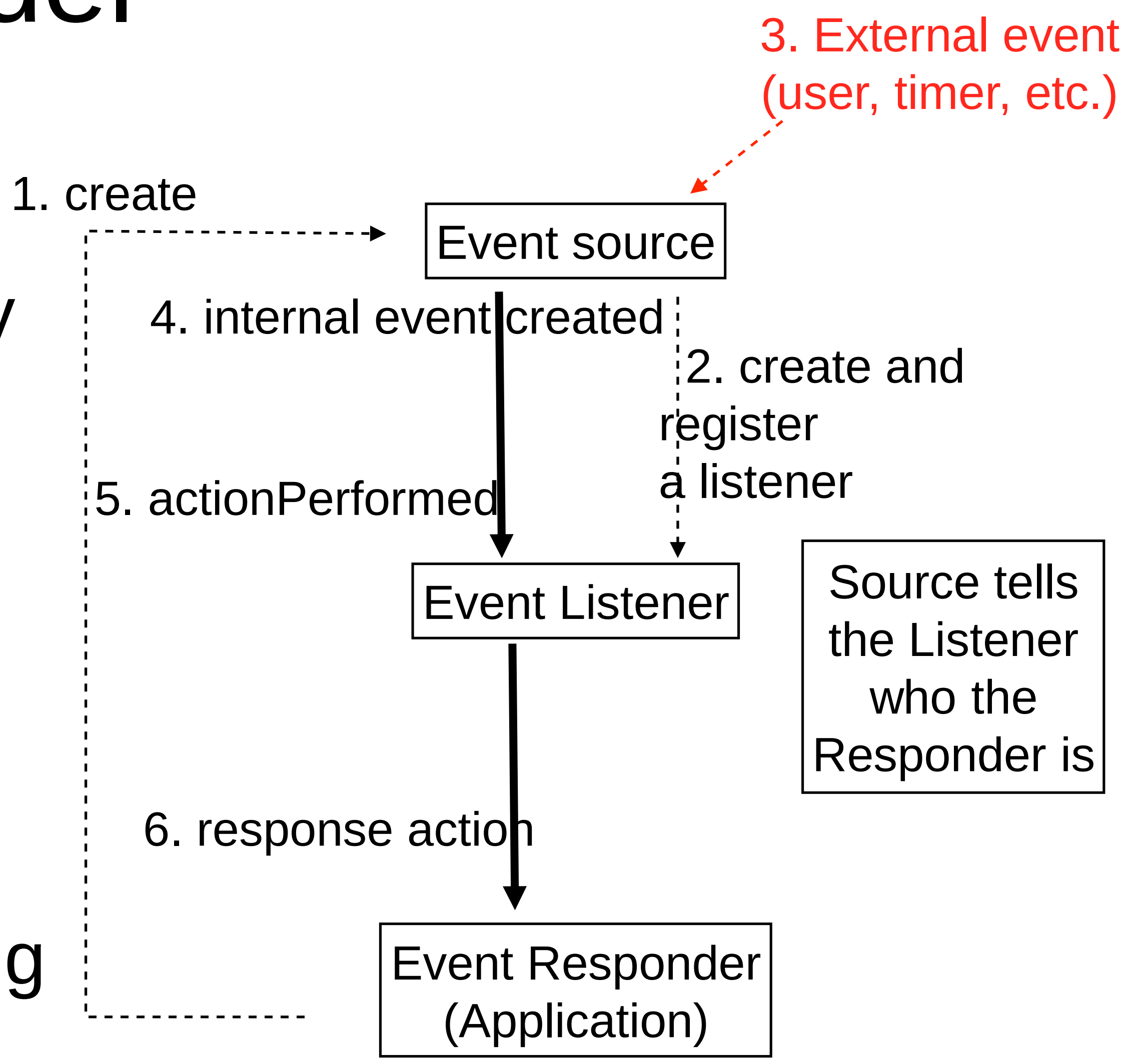
- User presses a mouse button
- User moves the mouse
- User opens or closes a window
- A clock tick occurs
- A *Timer* object’s time interval elapses

Event Handling

- Events happen
  - JRE (Java Runtime Environment) knows about them
- Programs want to know about events
  - but not all events and not all the time
- How does program tell JRE which events it cares about?
- How does JRE deliver those events to the program?

Java Event Model

- A Java Event is generated by an event *source* object
- An event *listener* object is “registered” with a source, which invokes the listener when an event occurs
- An application object is a *responder* to the event; the app creates a *source*, passing the responder object as an argument



3. External event (user, timer, etc.)

1. create

Event source

4. internal event created

2. create and register a listener

5. actionPerformed

Event Listener

Source tells the Listener who the Responder is

6. response action

Event Responder (Application)

Steps 1-2 are usually done once.
Steps 3-6 are usually repeated often.

Timer Event Example

When timer interval elapses, listener's actionPerformed is called

```
public class MoveTimer extends
                         javax.swing.Timer
{
  public MoveTimer( int t, Mover m )
  { ... _mover = m;...}
  private class MoveListener implements
               java.awt.event.ActionListener
  {
     public void actionPerformed( Event e )
     {
        _mover.move();
     }
     . . .
```

Call responder's <u>move</u> method

- *MoveTimer* is the event source
- *MoveListener* is the event listener
- *BallPanel* is the event responder

```
public class BallPanel implements Mover
{
  public BallPanel( ... )
  {
    MoveTimer = new MoveTimer( 100, this );
  }
  public void move() { . . . }
}
```

Create event source; tell it about responder (this)

MoveTimer

```
public class MoveTimer extends javax.swing.Timer
{
  Mover _mover;
  public MoveTimer( int interval, Mover m )
  {
    super( interval, null );
    _mover = m;
    this.addActionListener( new MoveListener() );
  }
  private class MoveListener implements
                    java.awt.event.ActionListener
  {
     public void actionPerformed( Event e )
     {
        _mover.move();
     }
  }
}
```

Register the ActionListener with the Timer class object.

interval is in milliseconds the *Mover* is the event responder object

ActionListener interface only has one method.

An inner class has access to instance variables of the outer class

MoveListener is an inner class and *private*

BallPanel

```
public class BallPanel extends JPanel implements Mover
{
  AEllipse _ball;  // book uses SmartEllipse,
                   // could also use JEllipse
  public BallPanel( )
  {
    _ball = new BouncingBall( Color.RED, this );
    Timer timer = new MoveTimer( 100, this );
    _timer.start();
  }
  ...
  public void move()
  {
    _ball.move();
    repaint();
  }
}
```

Create the timer and start it. The timer keeps generating events every 100 msecs until program issues a *timer.stop()* method invocation.

move() gets called by the *MoveListener* object after the time interval has elapsed; the Timer immediately starts up another interval countdown.

BouncingBall

```
public class BouncingBall extends AEllipse implements Mover
{ ...
  public BouncingBall( Color c, Container parent )
  {
    super( frame, aColor );
    _dX = 5; _dY = 5;
    _f = frame;
  } ...
  public void move()
  {
    int nextX = this.getX() + _dX; // update position
    int nextY = this.getY() + _dY;
    if ( nextX < 0 )
    {
      _dX = -_dx;      // if so, reverse x direction
      nextX = 0;
    }
    else if ( nextX + this.getWidth() >= _parent.getWidth())
    {
      _dX = -_dx;      // if so, reverse x direction
      nextX = _parent.getWidth()- this.getWidth();
    }
    ... // test top and bottom bounds
}
```

Book version extends *SmartEllipse*. Our *AEllipse* works also, with minor edits to book code

Compute next position

Is it to left of drawing area?

Is it to right of drawing area?

Note: Lab version uses ***Animated*** interface instead of *Mover* and ***FrameTimer*** class instead of *Timer*, but they are functionally equivalent

FrameTimer class *NewFrame* interface *Animated* interface

- We define variations of the *Timer* class and *Mover* interface:
  - *FrameTimer* class represents timer events that signal that a new frame should be created
  - *NewFrame* interface defines a general response to a new frame event, which need not be an animation response
  - *Animated* interface defines a general response for frame-by-frame animation events such that each object’s response to a frame event can be dynamically and independently enabled and disabled

NewFrame interface *Animated* interface

- *NewFrame* only implies that a new frame is required

```
public interface NewFrame
{
   public void newFrame();
}
```

- *Animated* implies animation may occur over a sequence of new frame events and the animation can be enabled/disabled for each object

```
public interface Animated extends NewFrame
{
   public void setAnimated( boolean onOff );
   public boolean isAnimated();
}
```

Moving Composite Objects

- There are 2 kinds of composite objects we can create:
  - *JComponents* that have their own origin (Lab 3 *JPlayer*)
  - Anything else, whose component coordinates are relative to the drawing panel (*A-classes* and *awt Graphics2D* objects)
- *JComponent* versions are trivial

```
public class JPlayer
extends JComponent
{
  public void newFrame()
  {
     // computes next position
      . . .
     this.setLocation( nextX, nextY );
     repaint();
  }
}
```

No need to override the *JComponent setLocation* method; all components are already defined relative to the *JComponent* location.

Other Composite Objects

- For a composite object that is not a *JComponent* (*SnowMan*)
  - *newFrame()* delegates the action to the components using their *moveBy* method
  - could also use *setLocation* method of components, but *moveBy* is simpler for composite and also useful for dragging.

newFrame() computes new position

```
public class SnowMan implements Animated
{
  public void newFrame()
  {
     // computes next position
      . . .
     this.setLocation( newX, newY );
  }
  public void setLocation( int x, int y )
  {
    int dx = x - this.getX();
    int dy = y - this.getY();

    head.moveBy( dx, dy );
    rightArm.moveBy( dx, dy );
    leftArm.moveBy( dx, dy );
    rightEye.moveBy( dx, dy );
      ...
    super.setLocation( x, y );
  }
}
```

setLocation gets each component to move itself to its new position.

Draggable interface

- Our *Draggable* interface expands the *wheels* version
  - it allows dragging to be enabled/disabled for each object
  - it requires a *moveBy* method to simplify re-positioning
  - it requires a *contains(Point2D)* method so a container can pass along its mouse events to any of its components that want to be draggable.

```
public interface Draggable
{
   public void setDraggable( boolean onOff );
   public boolean isDraggable();
   public boolean contains( Point2D p );
   public void moveBy( int x, int y );

}
```

Review

- Java Event handling model
  - Source, listener, responder
- *javax.swing.Timer* provides framework for animation
- Can animate *JComponents* (as in *JWheels*) as well as the *Graphics* objects (as in *AEllipse*, *et al.*)

Next, in 416

- User Interface Design
- More Swing features
- Read Chapter 8