SIENAcollege
Computer Science

Computer Science 225
Advanced Programming
Siena College
Spring 2020

Topic Notes: Event-Driven Programming

We have already looked at some *event-driven programs* as we worked with Java Swing components and Java graphics. We added *listeners* to our programs so methods called *event handlers* would be called when the user of the program did something like pressing a `JButton`, changing the value in a `JSlider` or `JComboBox`, selecting or unselecting a `JCheckbox` or `JRadioButton`, or performing some action with the mouse over a graphics window.

Now, we will consider how to make our GUI and graphics programs more interesting and responsive. Along the way, we will think about how we can use Java's object-oriented programming features to make our programs simpler, with more reusable code.

Maintaining a List of Items to Draw

A key idea to keep in mind when using Java graphics is that any time a change is made (often in response to an event), the `paintComponent` method of the component in which we are drawing must be redrawn in its entirety. We cannot simply draw one new graphics primitive, or erase values. This means we need to remember all of the information in persistent memory (*i.e.*, instance variables) that will allow the `paintComponent` method to draw the entire state of the program's graphics window.

As a first example of this, we have the `MouseDroppings` example. In this program, we draw a small circle (a "mouse dropping") every time the mouse moves within the graphics window, centered at the location where the mouse just moved. When the mouse exits the window, it is cleared.

A few things are needed to implement this:

1. We use a `JPanel` that overrides the `paintComponent` method as our graphics area.
2. We need two mouse event handlers: `mouseMoved` and `mouseExited`. These are required by two different mouse event handling interfaces, so to avoid the need to write all of the methods, we extend the `MouseAdapter` class and override only those two.
3. We maintain an instance variable which refers to a list of `Point` objects, which contains all of the mouse locations where a mouse moved event has occurred since the program started or the last time the mouse exited the window. This is used in a few places:
    - In the `mouseMoved` method, we retrieve the coordinates of the mouse pointer from the `MouseEvent` object using its `getPoint` method, and add that to the list.
    - In the `mouseExited` method, we clear the list.

- In the `redraw` method, which is called from `paintComponent`, we traverse the list of points and draws a filled oval centered at each position.

In our next example, `SpiralLines`, we draw a new line as the mouse is dragged, from the press point to the current mouse location during each drag event.

This is very similar to `MouseDroppings`. Here, we need to maintain a list of pairs of points where the lines need to be drawn in `paintComponent`. Instead of `mouseMoved`, we use `mousePressed` and `mouseDragged` event handlers.

Experiment: what happens if we update the `pressPoint` as well on each mouse drag?

Moving/Dragging Objects

In the `SunAndMoon` example, mouse actions control a celestial body that rises as the sun and sets as the moon. It's not a very accurate representation of the behavior of the actual sun and moon, but it does illustrate a few new things:

1. We track the position of the sun/moon with a single instance variable that is updated during mouse move/drag events, and is used in `paintComponent` to draw it in the correct location.
2. The three items drawn on the screen are done in a specific order: first the sky, then the sun/moon, and finally the ground. This ensures that the sun/moon appears on the screen "in front of" the sky, but "behind" the ground.

Generally when we think about a mouse drag operation, it involves dragging an icon or other object on our screens.

The example `UglyDragABall` does exactly this for a circle displayed in a graphics window.

A few key points here:

1. There is only one object to draw, and we store the coordinates where it should be in the `upperLeft` variable.
2. We only want to drag the ball if the mouse was initially pressed within the bounds of the circle, but the `mouseDragged` method would be called during a drag either way. The `dragging` variable remembers whether the initial mouse press was within the circle, and `mouseDragged` only moves the circle if it was.
3. During the drag operation an *absolute move* is performed, placing the upper left corner of the circle's bounding box at the current mouse location.

The problem with this example is that the upper left corner of the circle's bounding box follows the mouse pointer during a drag event. This results in an ugly "jump" at the start of the dragging, especially noticeable when the initial press point is far from the upper left.

In the improved example, `DragABall`, the dragging works like we would hope: as if we grabbed the circle at the mouse press point, and as we drag around, that point within the circle follows the mouse pointer.

What's changed?

- The `dragging` variable has been replaced with a `lastMouse` variable that remembers the last mouse press or drag position during a dragging operation. It is `null` if the last press was not within the circle.
- When updating the position of the upper left corner of the circle's bounding box, we *translate* the point, which is a *relative move* by the amount the mouse has moved since either the initial press or the last drag event.

For an in-class exercise, you enhanced the `DragABall` example to drag around two of them. In this next example, `DragMany`, we drag around many shapes. Some are circles, some are squares, some are filled, some are just an outline, and each is a different size and color.

There are several things to note about this example:

- Starting at the end, the `main` method is different from those in our previous Swing/graphics examples. This one takes a command-line parameter, the number of shapes we'll be drawing and dragging, parses it into an `int`, and passes it as a parameter to the `DragMany` constructor. Of course, this means we have to define a constructor in `DragMany`, something we did not have to do previously, as the default (0-parameter) constructors were sufficient.
- Our `DragMany` class also has a few named constants. This is a good habit to get into. As graphics programs get more and more complex, it's good to avoid having "magic numbers" for the positions and sizes of things, whose meaning is not necessarily obvious, sprinkled throughout our programs. Using the named constants brings a couple of benefits. First, when you're reading or modifying your code, you'll have a hint (assuming a meaningful name) about what the value's purpose is. Second, if you decide to change some of them, you can change them once where they are declared, and the new values will propagate through.
- Of course we need to maintain a list of the objects we'll be drawing and dragging, so our mouse event handlers can tell if we have pressed in one of them or not, and so our `paintComponent` method can draw them all. Since our objects can have a variety of characteristics, this list contains instances of a new custom class that defines all of the necessary properties. We'll look at those details soon.
- Given that all of the objects to draw are in that list, the `paintComponent` method is reduced to looping over the list and painting each.
- Our custom class `DraggableShape` provides all of the functionality we need to have shapes with the various characteristics that can be drawn and dragged around. This means we need to be able to specify all of these characteristics to the constructor, and the objects

need to be able to draw themselves on a given `Graphics` object (the `paint` method), perform a relative move (the `translate` method), and determine if a given point is within the bounds of the object (the `contains` method).

- The shapes are constructed added to the list in the `run` method, so they're in the list before the first paint event causes the first call to `paintComponent`. Note that the random numbers chosen for the initial positions are guaranteed to have the object on the screen.
- The same event handlers are needed here as in the other dragging examples.
  - In `mousePressed`, we give `lastMouse` a non-`null` value if there is an object being dragged, but now we also need to know which object it is that is being dragged. A reference to that object is stored in the `dragging` variable. Notice that we search for an object that contains the press point from the end of the list to the beginning, since it makes the most sense from a usability point of view to "pick up" the topmost shape when they overlap, and the shapes later in the list are drawn last, so appear to be "on top". Further, when we find the object we will be dragging, we `remove` and then re-`add` the object so it will then be last in the list, and hence drawn on top.
  - The `mouseDragged` and `mouseReleased` methods here are very similar to those in the earlier examples. However, the method call to `translate` is now calling the method we wrote in the `DraggableShape` custom class, not in a `Point`.