Lab 7 — Animation and Event-Driven Programming

1. Download and unzip **lab07_files.zip** from the course web page. Open **Aquarium**, **AquariumPanel**, **MovingImage**, and **BallFish** in DrJava, and examine the code. An Aquarium contains an AquariumPanel, which keeps track of a number of MovingImage objects in an internal ArrayList. When a new BallFish creature is added to the Aquarium by calling the Aquarium's `addCreature` method, it gets added to the AquariumPanel's internal ArrayList, and its `randomize` method is called (which currently does nothing).

   If you compile and run Aquarium, you'll notice that nothing happens. We first need to create a Timer and attach an appropriate listener object to it that will respond to Timer events. The AquariumPanel's inner **AquariumListener** class, which implements the ActionListener interface, is what we need. It defines the `actionPerformed` method, which responds to Timer events by printing "tick!" each time an event occurs.

   Add the following code to the AquariumPanel constructor to create a listener, attach it to a Timer, and start the Timer ticking every 1000 milliseconds. After recompiling AquariumPanel, when you run Aquarium you should now see "tick!" printed out once every second.

```
AquariumListener listener = new AquariumListener();
Timer t = new Timer(1000, listener);
t.start();
```

2. We still can't see any creatures, because the AquariumPanel's `paintComponent` method currently doesn't do anything. Instead, it needs to call each creature's `draw` method using a for-loop, like this:

```
for (MovingImage creature : this.allCreatures) {
    creature.draw(pen);
}
```

   Add this code to `paintComponent`, recompile AquariumPanel, and then rerun Aquarium. You should see a blue BallFish creature appear in the upper left corner of the aquarium, although it still doesn't move.

3. We want our AquariumListener to move the creatures on each tick, instead of printing a message. Because it is an inner class, it has direct access to the AquariumPanel's ArrayList of creatures, so it can update each creature's coordinates directly, and then redraw the AquariumPanel. Replace the print statement in the `actionPerformed` method with the following code:

```
for (MovingImage creature : allCreatures) {
    creature.update(getWidth(), getHeight());
}
repaint(); // schedules a call to paintComponent
```

   This version of `actionPerformed` calls the `update` method of each MovingImage in the ArrayList, passing in the width and height of the AquariumPanel as parameters. After all images have been updated, the `repaint()` method is called, which causes the AquariumPanel's `paintComponent` method to be invoked (remember that you can't call `paintComponent` yourself; think of `repaint` as a way of asking the system to call it for you).

   Recompile AquariumPanel and rerun Aquarium. On each timer tick, the AquariumPanel's action listener calls the BallFish's `update` method and then repaints the window, but the `update` method currently doesn't do anything, so the ball still just sits there. To move the ball, we need to update its *x, y* coordinates on each tick. Add the lines below to the `update` method and recompile BallFish, then rerun the code (and watch closely!):

```
this.x += 1;   // same as this.x = this.x + 1
this.y += 1;
```

   For better animation, we need to speed up our timer. Change the timer delay from 1000 milliseconds to 50 milliseconds (or even 10) and try again. Now you should see the ball move smoothly across the window.

4. Experiment with different effects, like incrementing *x* or *y* by an amount greater than 1 to increase the speed of the ball in the horizontal or vertical direction, or adding 1 to the radius on each tick.

5. Next, add a red BallFish creature to the Aquarium, in addition to the blue one. Why is only one creature visible? To fix this, we need to randomize the position and speed of each creature. Add some code to the BallFish `randomize` method that sets a creature's *x* and *y* coordinates to appropriate random values, so that creatures start out in different locations in the graphics window. Try to place the creature so that it remains fully visible within the drawing window. Also define new BallFish instance variables `dx` and `dy` to keep track of the creature's speed in the *x* and *y* directions, and have `randomize` set them to random values from –5 to +5. Here is a useful helper method for generating random integers from *start* to *end* – 1 (like ranges in Python):

```
public int randRange(int start, int end) {
    Random generator = new Random();
    return generator.nextInt(end - start) + start;
}
```

6. Now modify the code so that a ball will stay within the window by bouncing off the edges. When the AquariumPanel's action listener calls `update`, it passes in the width and height of the panel. Use these values to help you determine when to change the ball's direction. When the ball goes beyond the left or right edge of the window, negate `dx`. When it goes beyond the top or bottom edge, negate `dy`. See if you can make the ball reverse direction as soon as it *touches* an edge of the window, rather than waiting until its *center* meets the edge.

7. Now make the ball change color whenever it bounces off a wall. You can create random colors by calling `new Color(r, g, b)` with random values from 0-255 for `r`, `g`, and `b`, or you could create an internal array of Color objects and pick one at random from the array each time.

8. Take your aquatic-creature-drawing code from the last homework assignment and use it to create a new version of your Fish class that implements the MovingImage interface and moves your Fish around in some fashion, like the BallFish class. Then add your Fish to the Aquarium.

9. Open the files **MouseDemo** and **MousePanel** and examine the code. This program is set up to respond to mouse input using two types of listeners: one that implements the MouseListener interface, and another that implements the MouseMotionListener interface. All methods required for each interface are shown, but are currently empty. Compile and run the code to see what it does.

10. To understand what the `mousePressed` method of the MouseListener interface does (as implemented by the ListenerForEvents class), add the following line to the method and then recompile and test the code. This line simply changes the background color of the MousePanel to a new random color:

```
setBackground(randomColor());
```

11. Compare this behavior to that of `mouseReleased` and `mouseClicked` by moving the above line to each of the latter methods in turn and retesting the code. Compare the behavior of `mouseEntered` and `mouseExited` in the same way. Also try putting the line in both of those methods.

12. Compare the MouseMotionListener methods `mouseDragged` and `mouseMoved` in the same way.

13. To make the red ball respond to the mouse, we need to have the appropriate listener method(s) update the coordinate of the ball based on the location of the mouse event, and then repaint the MousePanel. We can find out the *x* and *y* coordinates of the mouse event from the MouseEvent object by calling `event.getX()` and `event.getY()`. Add some code to `mousePressed` that does this and try it out. Don't forget to call `repaint()`. Then try moving the code to the other methods in turn and compare the resulting behaviors.

14. Now let's make our program respond to keyboard events, such as pressing an arrow key or typing a character. We need to create a third listener class, which implements the KeyListener interface. This interface requires the following methods:

```
public void keyPressed(KeyEvent event)

public void keyReleased(KeyEvent event)

public void keyTyped(KeyEvent event)
```

Define a new listener class called ListenerForKeys, similar to ListenerForEvents and ListenerForMotion, containing the above methods. For now, leave `keyPressed` and `keyReleased` empty, but add the following code to `keyTyped`:

```
if (event.getKeyChar() == 'C') {
    setBackground(randomColor());
}
```

We want the background color to change whenever an uppercase C is typed. In addition to attaching a ListenerForKeys object to the panel, we also need to make the panel "focusable", because only focused components can receive key events. So we need the following two lines in the MousePanel constructor:

```
this.addKeyListener(new ListenerForKeys());
this.setFocusable(true);
```

After making these changes, compile and test your code.

15. What about non-printable keys such as the arrow keys? For those, we can have `keyPressed` or `keyReleased` test the event's *keycode* instead of the key character, using the *event*`.getKeyCode()` method. A set of special defined constants in the KeyEvent class called `VK_UP`, `VK_DOWN`, `VK_LEFT`, and `VK_RIGHT` are used to represent the arrow keycodes. The online documentation for the KeyEvent class lists many others. For example, to move the ball to the right whenever the right arrow key is pressed, add the following code to the `keyPressed` method:

```
if (event.getKeyCode() == KeyEvent.VK_RIGHT) {
    x = x + 10;
    repaint();
}
```

Add similar code for the other arrow keys, so that you can control the motion of the red ball via the keyboard.

16. Create an interactive drawing program called **MyDraw** that draws line segments using the arrow keys. The line should start from the center of the frame and extend toward the east, north, west, or south when the right-arrow key, up-arrow key, left-arrow key, or down-arrow key is pressed (respectively). You could also have the mouse specify points if you like. You'll need to store each point or line segment internally, perhaps using an ArrayList, so that whenever `paintComponent` is called, it will be able to reconstruct the drawing.

17. Enhance your drawing program by adding at least one new interesting feature to it, beyond the basic line-drawing functionality described above. For example, you could have your program respond to particular keys in different ways. Typing a C might put the program in "circle mode", so that clicking the mouse causes a circle to appear on the screen at the specified position, while typing an L returns the program to line-drawing mode. (Hint: if you use an instance variable to keep track of the current "mode", your listeners can test this variable to determine how to respond appropriately to an event.) Or maybe typing S scrambles all geometric objects currently on the screen by moving them to new random positions. Or maybe typing U or D moves all objects up or down by a small amount. There are many possibilities, so use your imagination. In any case, be sure to include some documentation in the form of comments at the top of your MyDraw program that clearly explain its features.