6. Getting observer response

In previous chapters we only displayed images and text on the screen. But in a psychophysics experiment you often want to get the observer's response to the stimuli presented, for instance through a key press. In this chapter I will show how to collect observer response using tools of Java. I will first explain the event handling mechanism in Java and demonstrate a sample implementation (an asymmetric matching experiment). This example emphasizes writing your own specialized event handling mechanism for the particular experiment you are implementing. Later I will show how to implement a thread safe mechanism to collect observer response into the FullScreen class. I will introduce generalized methods to conveniently collect observer response without writing a specialized event handler.

6.1. Event handling mechanism in Java

Events - such as key presses - are handeled by your Java program in the following manner: You first construct an *event source.* Event source can be a button, a menu or a window, including a FullScreen window. When an interesting activity occurs, for instance when a key is pressed or the mouse moves, the operating system informs your event source. On the other hand, the event source also keeps in touch with a set of *event listener objects* whose appropriate methods are invoked when an event occurs. Once an intresting event occurs, the event source creates an *event object,* which stores the information about the event. The event source then passes the event object to the appropriate method of the event listener object. This type of mechanism is often referred as *callback* mechanism.

Let's have a closer look at the components of this mechanism: First we need to construct an event source. Of course a FullScreen object, or any class inheriting from it, *is* an event source. So by creating an instance of FullScreen class you are automatically creating your event source. Next an event listener must be *added* to, i.e. associated with, the event source by invoking an appropriate method. For example suppose we want to record the observer's responses through the keyboard, then we must assign an *event listener* as follows

```
public class MyKeySource extends FullScreen1 {
  //...
  MyKeySource(){
    super();
    MyKeyListener mkl = new MyKeyListener();
    mks.addKeyListener(mkl);
  }
  //...
}
```

here mkl is the associated object that will listen to and trap the key events. The KeyListener object constitutes the other component of the event handling mechanism. MyKeyListener class should implement the KeyListener interface and *must* have the following three methods

```
class MyKeyListener implements KeyListener {
  public void keyPressed(KeyEvent ke) {
```

```
    // ..
  }
  public void keyReleased(KeyEvent ke) {
    // ..
  }
  public void keyTyped(KeyEvent ke) {
    // ..
  }
}
```

Whenever the observer hits a key, MyKeySource creates a KeyEvent object and invokes the proper method of MyKeyListener - one of the keyPressed(), keyReleased() or keyTyped() methods. You put the code determining the behavior of your program inside those methods.

This approach has one drawback: MyKeyListener is a class by itself and instances of it won't have access to fields of MyKeySource. One often needs the methods of the listener be able to access and modify properties of the source, for instance, typing in a letter could change the background color of the screen. There are several ways to mitigate this problem. One common approach is using an *annonymous inner class* (See Horstmann & Cornell, Chapter XXX)

```
public class MyKeySource extends FullScreen2 {
  public MyKeySource() {
    super();
    addKeyListener(new KeyListener(){
      public void keyPressed(KeyEvent ke) {
        // ..
      }
      public void keyReleased(KeyEvent ke) {
        // ..
      }
      public void keyTyped(KeyEvent ke) {
        // ..
      }
    });
  }
  //...
}
```

Now the methods of the KeyListener have access to the fields and methods of the outer class. This approach is practical, though it has its own drawbacks.

The approach I am going to adapt is different than the obove two. Recall that listener object that MyKeySource is in touch *must* implement the KeyListener interface, this is the only requirement. So we can make MyKeySource class implement KeyListener interface with its 3 necessary methods, then add *itself* as a listener, in other words, MyKeySource becomes both the source and listener of KeyEvents. Here is how that modified MyKeySourceAndListener class looks like

```
public class MyKeySourceAndListener extends FullScreen
    implements KeyListener {

  public MyKeySourceAndListener() {
```

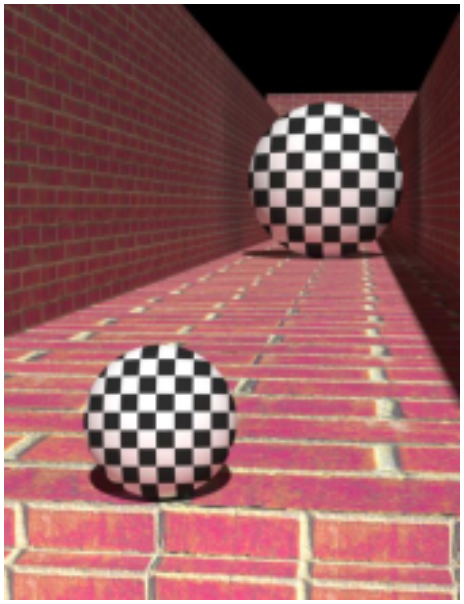

Figure 6.1.: The further sphere seems to occupy a portion of the visual field which is larger than the closer one although the visual angle they subtend are exactly the same.

```
      super();
      // ...
      addKeyListener(this);
    }
    // ...
    public void keyPressed(KeyEvent ke) {
      // ..
    }
    public void keyReleased(KeyEvent ke) {
      // ..
    }
    public void keyTyped(KeyEvent ke) {
      // ..
    }
    //...
  }
```

In the next section I will show how to implement this approach.

6.2. Writing your own specialized event handler

An object seems to subtend a larger visual angle than an other one subtending identical visual if its perceived distance to the observer is larger than the later one. Using two dimensional depth cues (such as percpective) one can create interesting visual illusions. One such illusion is illustrated in Figure 6.1. The sample example of this section will test the magnitude of illusion behaviorly using an asymmatric matching procedure (See Murray, Boyaci, and Kersten, Nat. Neuro. 2006).

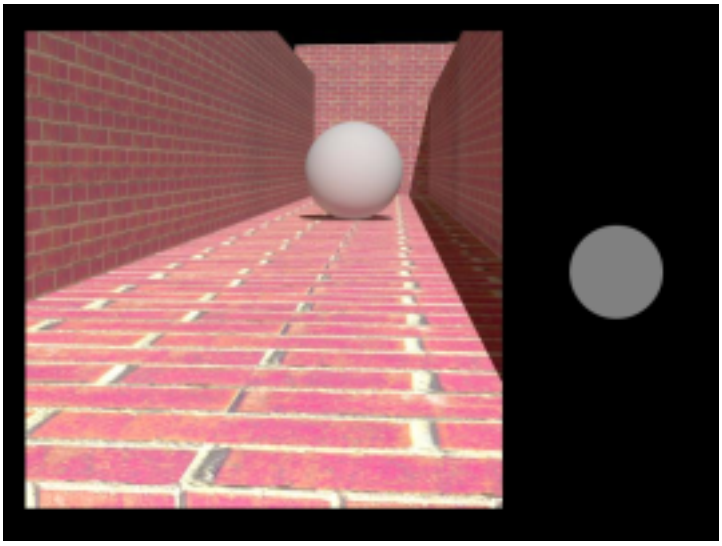

Figure 6.2.: Screen shot from the Sizes experiment.

In this sample experiment observers task is adjusting the size of a two dimensional disk until it matches the image size of a three dimensional ball in context (see Figure 6.2). Observer will adjust the size of the disk using the arrow keys and finalize each trial by pressing the space bar.

As I indicated before, you have a class which inherits from FullScreen and implements KeyListener interface, and then adds itself as a KeyListener

```
public class SizesAsM extends FullScreen1 implements Runnable, KeyListener {
  SizesAsM(){

    super();
    addKeyListener(this);
  }
```

As we have always been doing, SizesAsM extends FullScreen1 and implements Runnable. The main() method of the class is not very different from previous examples. Inside the run() method the program first reads in the necessary image files and determines the coordinates to place them. The program also creates a MAX_DISK_SIZE by MAX_DISK_SIZE BufferedImage to draw the matching disk.

The program then displays the instruction text on the screen and tells the observer to press the 's' key

```
displayText(100, 800, "Now hit the \"s\" key to start");
```

and waits until the observer actually presses the 's' key

```
while(!start){}
```

where start is a global boolean variable which is set to false initially. The keyTyped() method (remember that keyTyped() is one of the methods of the KeyListener interface which is implemented by SizesAsM) is responsable to change the value of start to true

```
public void keyTyped(KeyEvent ke) {

  char keyTypedChar = ke.getKeyChar();
  if (keyTypedChar == 's')
      start = true;
}
```

The method first gets the typed key as a character and if it is 's' it sets the value of the global variable start to true, which allows the experiment start.

In an asymmetric matching experiment you usually want to randomize the order of the test stimulus presented. Here is how you can do it. You first create an ArrayList with an initial capacity of total number of test stimuli

```
List<Integer> trials = new ArrayList<Integer>(N_BACK+N_FRONT);
```

Next you populate this ArrayList with integers

```
for (int i = 0; i < N_BACK + N_FRONT; i++)
  trials.add(i);
```

each representing a trial that corresponds to a different stimulus. The static method shuffle() of Collections class randomizes the order of elements of a List

```
Collections.shuffle(trials);
```

Once the trials are randomized you can get the elements from the ArrayList efficiently. First convert the ArrayList into an Iteretor

```
Iterator<Integer> it = trials.listIterator();
```

Next get the next element from the iteretor until there are no more elements left, and set the next stimulus image, s, to present to the observer

```
while (it.hasNext()) {
  int nStim = it.next();
  s = stim[nStim];
  updateScene();

  // ....
}
```

s is a global BufferedImage variable and the method updateScene() takes care of putting the images on the screen

```
public void updateScene() {
  blankScreen();
  displayImage(bgX, bgY, stimBG);
  displayImage(x, y, s);
  displayImage(matchX, matchY, match);
  updateScreen();
}
```

6. Getting observer response

After the stimulus and the matching disk is displayed on the screen, the program waits for observer response until the trial ends with observer's hitting the space bar

```
while(!trialDone){}
```

trilaDone is set to false at the beginning of each trial. While the trialDone is false the current thread waits idle, but the thread of the KeyListener goes on working in the backgorund. The keyPressed() method takes care of interpreting the observer's key strikes and updating the scene accordingly

```
public void keyPressed(KeyEvent ke) {
  int keyPressedCode = ke.getKeyCode();
```

this stores the code of the pressed key in an integer. First the program checks if the observer pressed the escape key

```
  if (keyPressedCode == KeyEvent.VK_ESCAPE) {
    closeScreen();
    System.exit(0);
  }
```

If the experiment has already started, the program checks whether any of the up/down/left/right arrow or space bar is pressed and either sets the size of the disk or sets the value of trialDone to true

```
  else if (start) {
    switch (keyPressedCode) {
      case LARGER:
        matchSize = min(MAX_DISK_SIZE, matchSize + 5);
      break;
      case SMALLER:
        matchSize = max(0, matchSize - 5);
      break;
      case fLARGER:
        matchSize = min(MAX_DISK_SIZE, matchSize + 20);
      break;
      case fSMALLER:
        matchSize = max(0, matchSize - 20);
      break;
      case COMPLETE:
        trialDone = true;
      break;
    }
```

LARGER, SMALLER etc. are global integer variables corresponding to the codes of arrow keys and the space bar, and they are set during the initialization of the SizesAsM class. At the end, if the pressed key was one of the arrow keys the program generates a new matching disk and updates the scene

```
    if (!trialDone) {
      generateMatch();
      updateScene();
    }
```

generateMatch() simply draws a gray disk cnetered on the BufferedImage match, then updateScene() method displays that new disk on the right hand side of the test image.

Here is the entire code of SizesAsM

```
/*
 * chapter 6: SizesAsM.java
 *
 * Shows how to write a specialized event handler for a
 * specific task.
 *
 */
import static java.lang.Math.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import java.io.*;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
import javax.imageio.ImageIO;
public class SizesAsM extends FullScreen1 implements Runnable, KeyListener {
  final static int MAX_DISK_SIZE = 240;
  final static int N_BACK = 5;
  final static int N_FRONT = 5;
  final static int N_TRIALS = 10;
  final static RenderingHints HINTS = new RenderingHints(
      RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  BufferedImage match;
  Graphics2D gMatch;
  BufferedImage stimBG;
  int matchSize;
  int matchX;
  int matchY;
  int bgX;
  int bgY;
  int x;
  int y;
  BufferedImage s;
  boolean start = false;
  boolean trialDone = false;
  final static int LARGER = KeyEvent.VK_UP;
  final static int SMALLER = KeyEvent.VK_DOWN;
  final static int fLARGER = KeyEvent.VK_RIGHT;
  final static int fSMALLER = KeyEvent.VK_LEFT;
  final static int COMPLETE = KeyEvent.VK_SPACE;
  SizesAsM(){
```

```
  super();
  addKeyListener(this);
}

public void keyTyped(KeyEvent ke) {
  char keyTypedChar = ke.getKeyChar();
  if (keyTypedChar == 's')
    start = true;
}

public void keyPressed(KeyEvent ke) {
  int keyPressedCode = ke.getKeyCode();
  if (keyPressedCode == KeyEvent.VK_ESCAPE) {
    closeScreen();
    System.exit(0);
  }
  else if (start) {
    switch (keyPressedCode) {
      case LARGER:
        matchSize = min(MAX_DISK_SIZE, matchSize + 1);
      break;
      case SMALLER:
        matchSize = max(0, matchSize - 1);
      break;
      case fLARGER:
        matchSize = min(MAX_DISK_SIZE, matchSize + 20);
      break;
      case fSMALLER:
        matchSize = max(0, matchSize - 20);
      break;
      case COMPLETE:
        trialDone = true;
      break;
    }
    if (!trialDone) {
      generateMatch();
      updateScene();
    }
  }
}

public void keyReleased(KeyEvent ke) {

}

public static void main(String[] args) {

  SizesAsM sAM = new SizesAsM();
```

```
    sAM.setNBuffers(2);
    Thread experiment = new Thread(sAM);
    experiment.start();
  }
public void run() {
    try {
      stimBG = ImageIO.read(new File("bg.jpg"));
      bgX = (getWidth() / 2 - stimBG.getWidth()) / 2;
      bgY = (getHeight() - stimBG.getHeight()) / 2;
      BufferedImage[] stim = new BufferedImage[N_FRONT + N_BACK];
      for (int i = 0; i < N_FRONT; i++)
        stim[i] = ImageIO.read(new File("front_" + Integer.toString(i + 1)
            + ".jpg"));
      for (int i = 0; i < N_BACK; i++)
        stim[i + N_FRONT] = ImageIO.read(new File("back_"
            + Integer.toString(i + 1) + ".jpg"));
      int stimX_Back = bgX;
      int stimY_Back = bgY;
      int stimX_Front = bgX;
      int stimY_Front = bgY + (stimBG.getHeight() - stim[0].getHeight());
      match = (BufferedImage) createImage(MAX_DISK_SIZE, MAX_DISK_SIZE);
      matchX = getWidth() / 2 + (getWidth() / 2 - match.getWidth()) / 2;
      matchY = (getHeight() - match.getHeight()) / 2;
      List<Integer> trials = new ArrayList<Integer>(N_BACK + N_FRONT);
      for (int i = 0; i < N_BACK + N_FRONT; i++)
        trials.add(i);
      displayText(100, 200, "Task:");
      displayText(100, 350,
          "  Use the arrow keys to adjust the size of the disk on the right");
      displayText(100, 400,
          "  to match the image size of the ball in the scene");
      displayText(100, 450,
          "  (Up/Down for fine adjustment; Left/Right for fast adjustment)");
      displayText(100, 500, "  Press space bar to finalize trial");
      displayText(100, 600, "  Press ESC to quit");
      displayText(100, 800, "Now hit the \"s\" key to start");
      updateScreen();
      while (!start) {}
      for (int t = 0; t < N_TRIALS; t++) {
        Collections.shuffle(trials);
        Iterator<Integer> it = trials.listIterator();
        while (it.hasNext()) {
          trialDone = false;
          matchSize = (int) (random() * 150);
          generateMatch();
          int nStim = it.next();
          // for feedback: largest ball has 179 pixels, others are calculated
          // from their index
```

```
        String message = new String("Test Size = "
            + Integer.toString((nStim % 5 + 1) * 20 * 179 / 100) + ";");
        s = stim[nStim];
        if (nStim < N_FRONT) {
          x = stimX_Front;
          y = stimY_Front;
        }
        else {
          x = stimX_Back;
          y = stimY_Back;
        }
        updateScene();
        while (!trialDone) {}
        message = message
            + new String(" your response = " + matchSize + " (pixels)");
        blankScreen();
        // feedback message
        displayText(message);
        updateScreen();
        Thread.sleep(3000);
      }
    }
  } catch (IOException e) {
    System.err.println("File not found");
    e.printStackTrace();
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  finally {
    closeScreen();
  }
}

public void updateScene() {

  blankScreen();
  displayImage(bgX, bgY, stimBG);
  displayImage(x, y, s);
  displayImage(matchX, matchY, match);
  updateScreen();
}

public void generateMatch() {

  gMatch = match.createGraphics();
  gMatch.setRenderingHints(HINTS);
  gMatch.setColor(Color.BLACK);
```

```
    gMatch.fillRect(0, 0, MAX_DISK_SIZE, MAX_DISK_SIZE);
    int x = (int) floor((MAX_DISK_SIZE - matchSize) / 2);
    int y = x;
    gMatch.setColor(Color.GRAY);
    gMatch.fillOval(x, y, matchSize, matchSize);
    gMatch.dispose();
  }
}
```

6.3. A built-in thread safe event handler for the FullScreen class

In the previous section I showed how to write a specific event handler for an experiment. Even though that approach is usually the right one, it is not always very pratical. In certain situations it becomes harder and harder to implement the experiment in that way. Suppose, for example, that in different stages of the experiment hitting a certain key should result in different behaviour. Implementing this with the above approach would be difficult, at least would result in ugly looking and confusing code. Below I will show how to implement a built-in event handler in the FullScreen class, which makes the event handling in your experiment much easier and much more flexible.

Sooner or later one needs to write a threaded program (see Chapter XXX for threaded programming). In a threaded program it is essential to implement a thread safe event handling mechanism. For example, you wouldn't want different two different threads, one writing the other trying to read, accessing the key presses simultaneously. In a sequential program you don't need to worry about the access to the data fields, one method modifies a field, another reads it. No one steps on someone else's toe. But when a program runs on multiple threads, a method may try to modify a field while another method is reading it.

Below I will develop a thread safe event handling mechanism built into the FullScreen class, which allows you to easily and effortlessly collect observer responses. Here is how it works in principle: everytime an event occurs the appropriate methods of FullScreen (for example keyPressed() method) adds a new element to a thread safe BlockingQueue object. I will then implement methods which allow your program to access this BlockingQueue object and read the "head" (the first event element put to the BlockingQueue object) or reset ("flush") the entire event list.

I implement two BlockingQueue lists, one for the code of the event, another for the time when the event occured (this approach, rather than storing the event itself in a single list improves flexibility)

```
public class FullScreen2 extends JFrame impliments KeyListener {
  private BlockingQueue<Integer> keyPressed;
  private BlockingQueue<Long> whenKeyPressed;
  // ...

  FullScreen2(){
    // ...
    keyPressed = new LinkedBlockingQueue<Integer>();
    whenKeyPressed = new LinkedBlockingQueue<Long>();
  }

  //...
}
```

6. Getting observer response

We can now focus on handling the KeyEvents. Let's first decide on what to do in case the observer presses a key. We should, of course, determine which key is pressed, then the time of the key press

```
  public void keyPressed(KeyEvent ke) {

    if (ke.getKeyCode() == KeyEvent.VK_ESCAPE) {
      closeScreen();
      System.exit(0);
    }
    else {
      keyPressed.offer(ke.getKeyCode());
      whenKeyPressed.offer(ke.getWhen());
    }
  }
```

getKeyCode() method returns the *virtual key code* of the key pressed. For instance, if the up arrow key is pressed the returned value is VK_UP. getWhen() method returns the actual time when the event occured (in milliseconds). If the escape key is pressed the program terminates. The offer() method inserts the specified element into the LinkedBlockingQueue, returning true upon success, false if the capacity of the is exceeded. Similarly

```
public void keyReleased(KeyEvent ke) {
  try {
    keyReleased.offer(ke.getKeyCode());
    whenKeyReleased.offer(ke.getWhen());
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
}
```

returns the information about key releases. In case, for example, the observer presses "A" - capital a - FullScreen2 captures it in the keyTyped() method

```
public void keyTyped(KeyEvent ke) {
  try {
    keyTyped.offer(String.valueOf(ke.getKeyChar()));
    whenKeyTyped.offer(ke.getWhen());
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
}
```

here getKeyChar() method returns a char type data. Note that I convert the char into String because a BlockingQueue can hold only objects and char is not an object (it is a primitive and doesn't have an object wrapper as, for example, int does with Integer.)

Above three methods write into the BlockingQueue, next let's see the querry methods that your program should use to access the stored elements in those BlockingQueue

6. Getting observer response

```
public Integer getKeyPressed(long ms){

  Integer c = null;
  try {
    if(ms < 0)
      c = keyPressed.take();
    else
      c = keyPressed.poll(ms, TimeUnit.MILLISECONDS);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  return c;
}
```

poll() method retrieves the head of the BlockingQueue and also removes it from the queue. If invoked with a time variable, it waits for the specified amount of time until an element becomes available, if no element becomes available before the end of that time period it returns null. take() is similar to poll(), except it waits indefinetaly until an element becomes avialable. So if you invoke the getKeyPressed() with a negative argument, it will wait until your observer presses a key, on the other hand if you invoke it with a positive value (or zero), it will wait only for the amount of time (in milliseconds) that you specified for the observer response, if no element is available before the time limit it returns null. For convinience I overload the getKeyPressed() method

```
public Integer getKeyPressed(){

  return keyPressed.poll();
}
```

this overloaded version returns the head of the queue immediately, if the queue is empty it returns null.

```
public Long getWhenKeyPressed(){

  return whenKeyPressed.poll();
}
```

returns the time when the key press event occured and removes it from the queue. flushKeyPressed() method clears both keyPressed and whenKeyPressed queues

```
public void flushKeyPressed(){

  keyPressed.clear();
  whenKeyPressed.clear();
}
```

The entire listing of FullScreen2 class is given below

```
/*
 * chapter 6: FullScreen2.java
 *
 * Provides methods to get key presses
 *
 */
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.font.FontRenderContext;
import java.awt.geom.Rectangle2D;
import java.awt.image.BufferedImage;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import javax.swing.JFrame;
public class FullScreen2 extends JFrame implements KeyListener{
  private static final GraphicsEnvironment gEnvironment = GraphicsEnvironment
      .getLocalGraphicsEnvironment();
  private static final GraphicsDevice gDevice = gEnvironment
      .getDefaultScreenDevice();
  private static final GraphicsConfiguration gConfiguration = gDevice
      .getDefaultConfiguration();

  private int nBuffers = 1;
  private Color bgColor = Color.BLACK;
  private Color fgColor = Color.LIGHT_GRAY;
  private final Font defaultFont = new Font("SansSerif", Font.BOLD, 36);

  private BlockingQueue<String> keyTyped;
  private BlockingQueue<Long> whenKeyTyped;
  private BlockingQueue<Integer> keyPressed;
  private BlockingQueue<Long> whenKeyPressed;
  private BlockingQueue<Integer> keyReleased;
  private BlockingQueue<Long> whenKeyReleased;

  public void keyTyped(KeyEvent ke) {
    keyTyped.offer(String.valueOf(ke.getKeyChar()));
    whenKeyTyped.offer(ke.getWhen());
  }
  public void keyReleased(KeyEvent ke) {

    keyReleased.offer(ke.getKeyCode());
    whenKeyReleased.offer(ke.getWhen());
  }

  public void keyPressed(KeyEvent ke) {
```

```
  if (ke.getKeyCode() == KeyEvent.VK_ESCAPE) {
    closeScreen();
    System.exit(0);
  }
  else {
    keyPressed.offer(ke.getKeyCode());
    whenKeyPressed.offer(ke.getWhen());
  }
}

public FullScreen2() {
  super(gConfiguration);
  try {
    setUndecorated(true);
    setIgnoreRepaint(true);
    setResizable(false);
    setFont(defaultFont);
    setBackground(bgColor);
    setForeground(fgColor);
    gDevice.setFullScreenWindow(this);

    keyTyped = new LinkedBlockingQueue<String>();
    whenKeyTyped = new LinkedBlockingQueue<Long>();
    keyPressed = new LinkedBlockingQueue<Integer>();
    whenKeyPressed = new LinkedBlockingQueue<Long>();
    keyReleased = new LinkedBlockingQueue<Integer>();
    whenKeyReleased = new LinkedBlockingQueue<Long>();
    addKeyListener(this);

    setNBuffers(nBuffers);
  }
  finally {}
}
public void setNBuffers(int n) {
  try {
    createBufferStrategy(n);
    nBuffers = n;
  } catch (IllegalArgumentException e) {
    System.err.println(
        "Exception in FullScreen.setNBuffers(): "
        +"requested number of Buffers is illegal - falling back to default");
    createBufferStrategy(nBuffers);
  }
  try{
    Thread.sleep(200);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
```

```
}
public int getNBuffers() {
  return nBuffers;
}
public void updateScreen() {

  if (getBufferStrategy().contentsLost())
    setNBuffers(nBuffers);
  getBufferStrategy().show();
}

public void displayImage(BufferedImage bi) {
  if( bi!=null){
    double x = (getWidth() - bi.getWidth()) / 2;
    double y = (getHeight() - bi.getHeight()) / 2;
    displayImage((int) x, (int) y, bi);
  }
}
public void displayImage(int x, int y, BufferedImage bi) {
  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null)
      g.drawImage(bi, x, y, null);
  }
  finally {
    g.dispose();
  }
}
public void displayText(String text) {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  if( g!=null && text!= null){
    Font font = getFont();
    g.setFont(font);
    FontRenderContext context = g.getFontRenderContext();
    Rectangle2D bounds = font.getStringBounds(text, context);
    double x = (getWidth() - bounds.getWidth()) / 2;
    double y = (getHeight() - bounds.getHeight()) / 2;
    double ascent = -bounds.getY();
    double baseY = y + ascent;
    displayText((int) x, (int) baseY, text);
  }
  g.dispose();
}
public void displayText(int x, int y, String text) {
  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && text!=null){
      g.setFont(getFont());
```

```
      g.setColor(getForeground());
      g.drawString(text, x, y);
    }
  }
  finally {
    g.dispose();
  }
}
public void blankScreen() {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null){
      g.setColor(getBackground());
      g.fillRect(0, 0, getWidth(), getHeight());
    }
  }
  finally {
    g.dispose();
  }
}

public Color getBackground(){

  return bgColor;
}

public void setBackground(Color bg){

  bgColor = bg;
}

public void hideCursor() {
  Cursor noCursor = null;
  Toolkit tk = Toolkit.getDefaultToolkit();
  Dimension d = tk.getBestCursorSize(1,1);
  if((d.width|d.height)!=0)
    noCursor = tk.createCustomCursor(
        gConfiguration.createCompatibleImage(d.width, d.height),
        new Point(0, 0), "noCursor");
  setCursor(noCursor);
}

public void showCursor() {
  setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
}

public void closeScreen() {
  gDevice.setFullScreenWindow(null);
```

```
  dispose();
}

public String getKeyTyped(long ms){

  String c = null;
  try {
    if(ms < 0)
      c = keyTyped.take();
    else
      c = keyTyped.poll(ms, TimeUnit.MILLISECONDS);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  return c;
}

public String getKeyTyped(){

  return keyTyped.poll();
}

public Long getWhenKeyTyped(){

  return whenKeyTyped.poll();
}

public void flushKeyTyped(){

  keyTyped.clear();
  whenKeyTyped.clear();
}

public Integer getKeyPressed(long ms){

  Integer c = null;
  try {
    if(ms < 0)
      c = keyPressed.take();
    else
      c = keyPressed.poll(ms, TimeUnit.MILLISECONDS);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  return c;
}
```

```
  public Integer getKeyPressed(){

    return keyPressed.poll();
  }

  public Long getWhenKeyPressed(){

    return whenKeyPressed.poll();
  }

  public void flushKeyPressed(){

    keyPressed.clear();
    whenKeyPressed.clear();
  }

  public Integer getKeyReleased(long ms){

    Integer c = null;
    try {
      if(ms < 0 )
        c = keyReleased.take();
      else
        c = keyReleased.poll(ms, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      e.printStackTrace();
    }
    return c;
  }

  public Integer getKeyReleased(){

    return keyReleased.poll();
  }

  public Long getWhenKeyReleased(){

    return whenKeyReleased.poll();
  }

  public void flushKeyReleased(){

    keyReleased.clear();
    whenKeyReleased.clear();
  }
}
```

6.3.1. Examples using built-in methods

Here is the new version of the “Sizes” experiment using the built-in methods of the new FullScreen2 class

```
/*
 * chapter 6: SizesAsM2.java
 *
 * Shows how to use built-in methods of FullScreen for collecting
 * observer response
 *
 */

import static java.lang.Math.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import java.io.*;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
import javax.imageio.ImageIO;

public class SizesAsM2 extends FullScreen2 implements Runnable {

  final static int MAX_DISK_SIZE = 240;
  final static int N_BACK = 5;
  final static int N_FRONT = 5;
  final static int N_TRIALS = 10;

  final static RenderingHints HINTS = new RenderingHints(
      RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  BufferedImage match;
  Graphics2D gMatch;
  BufferedImage stimBG;

  int matchSize;
  int matchX;
  int matchY;
  int bgX;
  int bgY;

  int x;
  int y;
  BufferedImage s;

  final static int LARGER = KeyEvent.VK_UP;
  final static int SMALLER = KeyEvent.VK_DOWN;
  final static int fLARGER = KeyEvent.VK_RIGHT;
```

```
final static int fSMALLER = KeyEvent.VK_LEFT;
final static int COMPLETE = KeyEvent.VK_SPACE;

public static void main(String[] args) {

  SizesAsM2 sAM = new SizesAsM2();
  sAM.setNBuffers(2);
  Thread experiment = new Thread(sAM);
  experiment.start();
}

public void run() {

  try {
    stimBG = ImageIO.read(new File("bg.jpg"));
    bgX = (getWidth() / 2 - stimBG.getWidth()) / 2;
    bgY = (getHeight() - stimBG.getHeight()) / 2;
    BufferedImage[] stim = new BufferedImage[N_FRONT + N_BACK];
    for (int i = 0; i < N_FRONT; i++)
      stim[i] = ImageIO.read(new File("front_" + Integer.toString(i + 1)
          + ".jpg"));
    for (int i = 0; i < N_BACK; i++)
      stim[i + N_FRONT] = ImageIO.read(new File("back_"
          + Integer.toString(i + 1) + ".jpg"));
    int stimX_Back = bgX;
    int stimY_Back = bgY;
    int stimX_Front = bgX;
    int stimY_Front = bgY + (stimBG.getHeight() - stim[0].getHeight());
    match = (BufferedImage) createImage(MAX_DISK_SIZE, MAX_DISK_SIZE);
    matchX = getWidth() / 2 + (getWidth() / 2 - match.getWidth()) / 2;
    matchY = (getHeight() - match.getHeight()) / 2;

    List<Integer> trials = new ArrayList<Integer>(N_BACK + N_FRONT);
    for (int i = 0; i < N_BACK + N_FRONT; i++)
      trials.add(i);

    displayText(100, 200, "Task:");
    displayText(100, 350,
        "  Use the arrow keys to adjust the size of the disk on the right");
    displayText(100, 400,
        "  to match the image size of the ball in the scene");
    displayText(100, 450,
        "  (Up/Down for fine adjustment; Left/Right for fast adjustment)");
    displayText(100, 500, "  Press space bar to finalize trial");
    displayText(100, 600, "  Press ESC to quit");
    displayText(100, 800, "Now hit the \"s\" key to start");
    updateScreen();
```

6. Getting observer response

```
while(!getKeyTyped(-1).equals("s")){}

for (int t = 0; t < N_TRIALS; t++) {

  Collections.shuffle(trials);
  Iterator<Integer> it = trials.listIterator();

  while (it.hasNext()) {

    boolean trialDone = false;
    matchSize = (int) (random() * 150);
    generateMatch();
    int nStim = it.next();
    String message = new String("Test Size = "
        + Integer.toString((nStim % 5 + 1) * 20 * 179 / 100) + ";");
    s = stim[nStim];
    if (nStim < N_FRONT) {
      x = stimX_Front;
      y = stimY_Front;
    }
    else {
      x = stimX_Back;
      y = stimY_Back;
    }
    updateScene();
    flushKeyPressed();

    while (!trialDone) {

      Integer keyPressedCode = getKeyPressed(-1);
      switch (keyPressedCode) {
        case LARGER:
          matchSize = min(MAX_DISK_SIZE, matchSize + 2);
          break;
        case SMALLER:
          matchSize = max(0, matchSize - 2);
          break;
        case fLARGER:
          matchSize = min(MAX_DISK_SIZE, matchSize + 20);
          break;
        case fSMALLER:
          matchSize = max(0, matchSize - 20);
          break;
        case COMPLETE:
          trialDone = true;
          break;
      }
      if (!trialDone) {
```

```
                generateMatch();
                updateScene();
              }
            }
            message = message
                + new String(" your response = " + matchSize + " (pixels)");
            blankScreen();
            displayText(message);
            updateScreen();
            Thread.sleep(3000);
          }
        }
      } catch (IOException e) {
        System.err.println("File not found");
        e.printStackTrace();
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        e.printStackTrace();
      }
      finally {
        closeScreen();
      }
    }

    public void updateScene() {

      blankScreen();
      displayImage(bgX, bgY, stimBG);
      displayImage(x, y, s);
      displayImage(matchX, matchY, match);
      updateScreen();
    }

    public void generateMatch() {

      gMatch = match.createGraphics();
      gMatch.setRenderingHints(HINTS);
      gMatch.setColor(Color.BLACK);
      gMatch.fillRect(0, 0, MAX_DISK_SIZE, MAX_DISK_SIZE);
      int x = (int) floor((MAX_DISK_SIZE - matchSize) / 2);
      int y = x;
      gMatch.setColor(Color.GRAY);
      gMatch.fillOval(x, y, matchSize, matchSize);
      gMatch.dispose();
    }
  }
```

Here is another example which allows you to test the timing accuracy of key events

```
/*
 * chapter 6: KeyPressTest.java
 *
 * Test the observer response through keyboard
 *
 */
import java.awt.Font;
import static java.lang.Math.*;

public class KeyPressTest extends FullScreen2 implements Runnable {

  public static void main(String[] args) {

    KeyPressTest kpt = new KeyPressTest();
    kpt.setNBuffers(2);
    new Thread(kpt).start();
  }

  public void run(){

    try {
      displayText(100, 100, "Press keys on your keyboard during this test");
      displayText(100, 150, "Press q to finish the test");
      displayText(100, 200, "Now, press any key to start");
      updateScreen();
      getKeyPressed(-1);
      blankScreen();

      String res;
      int count = 0;
      while (true) {
        res = getKeyTyped(-1);
        blankScreen();
        displayText(count++ + " You pressed: " + res);
        updateScreen();
        if (res.equals("q"))
          break;
        flushKeyTyped();
      }
      flushKeyPressed();
      blankScreen();
      displayText(100, 100, "Next we test the reaction time");
      displayText(100, 150, "Press a key during the counting");
      displayText(100, 200, "Press q to finish the test");
      displayText(100, 250, "Now, press any key to start");
      updateScreen();
      getKeyPressed(-1);
```

```
      setFont(new Font("SansSerif", Font.BOLD, 46));
      while (true) {
        res = null;
        flushKeyTyped();
        count = 20;
        long start = System.currentTimeMillis();
        while (count>=0) {
          start = System.currentTimeMillis();
          blankScreen();
          displayText(300, 400, "counter: " + String.valueOf(count--));
          updateScreen();
          Thread.sleep(max(0,160-System.currentTimeMillis()+start));
        }
        res = getKeyTyped(1000);
        blankScreen();
        if(res != null){
          displayText(300, 600, "  You pressed: "
            + res + "   Reaction time: "
            + (getWhenKeyTyped() - start));
          if (res.equals("q"))
            break;
        }
        else
          displayText(300,600, "Time out! You are too slow!");
        updateScreen();
        Thread.sleep(3000);
      }
      Thread.sleep(1000);
    } catch (InterruptedException e) {}
    finally {
      closeScreen();
    }
  }
}
```

6.4. Summary

6.4.1. On using the FullScreen methods

To capture observer responses using the built-in methods of FullScreen class:

- Use getKeyTyped() method(s) to capture the characters the observer presses. For instance you can capture a capital a, “A”, with these methods (even though what the observer presses is possible “shift” and “a” keys together.)
- Use getKeyPressed() method(s) to get the code of the key hit by the observer. This method can report keys that don’t correspondond to ASCII characters such as arrow keys. Use getKeyPressed() methods if you need to capture such key presses. Similarly use getKeyReleased() method(s) to get the code of the released keys.

- Use getWhenKeyTyped(), getWhenKeyPressed() and getWhenKeyReleased() methods to get the time of the keyTyped, keyPressed and keyReleased events.
- getKeyTyped(), getKeyPressed() and getKeyReleased() methods are all overloaded. Invoke them with no argument if you need to immediately get the first key the observer has pressed. Those methods return null if the observer hasn't pressed anykey. Invoke those methods with a negative argument (for example with -1) if you want your program wait until the observer presses any key. Invoke them with a positive value (for example 1000) if you want to wait for the observer press a key for the time you specified in the argument (in milliseconds). If no key is pressed during the specified time the methods return null.
- use flushKeyTyped(), flushKeyPressed() and flushKeyReleased() methods to clear the event buffers.

6.4.2. On writing your own specialized event handler

...