```
/*************************************************************************
 *  Compilation:  javac StdDraw.java
 *  Execution:    java StdDraw
 *  Dependencies: none
 *
 *  Standard drawing library. This class provides a basic capability for
 *  creating drawings with your programs. It uses a simple graphics model that
 *  allows you to create drawings consisting of geometric shapes (e.g.,
 *  points, lines, circles, rectangles) in a window on your computer
 *  and to save the drawings to a file.
 *
 *  Todo
 *  ----
 *    -  Add support for gradient fill, etc.
 *    -  Fix setCanvasSize() so that it can be called only once.
 *    -  Should setCanvasSize() reset xScale(), yScale(), penRadius(),
 *       penColor(), and font()
 *    -  On some systems, drawing a line (or other shape) that extends way
 *       beyond canvas (e.g., to infinity) dimensions does not get drawn.
 *
 *  Remarks
 *  -------
 *    -  don't use AffineTransform for rescaling since it inverts
 *       images and strings
 *
 *************************************************************************/

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Component;
import java.awt.FileDialog;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.RenderingHints;
import java.awt.Toolkit;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

import java.awt.geom.Arc2D;
import java.awt.geom.Ellipse2D;
import java.awt.geom.GeneralPath;
import java.awt.geom.Line2D;
import java.awt.geom.Rectangle2D;

import java.awt.image.BufferedImage;
import java.awt.image.DirectColorModel;
import java.awt.image.WritableRaster;

import java.io.File;
import java.io.IOException;

import java.net.MalformedURLException;
import java.net.URL;
```

```
import java.util.LinkedList;
import java.util.TreeSet;
import java.util.NoSuchElementException;
import javax.imageio.ImageIO;

import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.KeyStroke;

/**
 *  The {@code StdDraw} class provides a basic capability for
 *  creating drawings with your programs. It uses a simple graphics model that
 *  allows you to create drawings consisting of points, lines, squares,
 *  circles, and other geometric shapes in a window on your computer and
 *  to save the drawings to a file. Standard drawing also includes
 *  facilities for text, color, pictures, and animation, along with
 *  user interaction via the keyboard and mouse.
 *  <p>
 *  <b>Getting started.</b>
 *  To use this class, you must have {@code StdDraw.class} in your
 *  Java classpath. If you used our autoinstaller, you should be all set.
 *  Otherwise, either download
 *  <a href = "https://introcs.cs.princeton.edu/java/code/stdlib.jar">stdlib.jar</a>
 *  and add to your Java classpath or download
 *  <a href = "https://introcs.cs.princeton.edu/java/stdlib/StdDraw.java">StdDraw.java</a>
 *  and put a copy in your working directory.
 *  <p>
 *  Now, type the following short program into your editor:
 *  <pre>
 *   public class TestStdDraw {
 *       public static void main(String[] args) {
 *           StdDraw.setPenRadius(0.05);
 *           StdDraw.setPenColor(StdDraw.BLUE);
 *           StdDraw.point(0.5, 0.5);
 *           StdDraw.setPenColor(StdDraw.MAGENTA);
 *           StdDraw.line(0.2, 0.2, 0.8, 0.2);
 *       }
 *   }
 *  </pre>
 *  If you compile and execute the program, you should see a window
 *  appear with a thick magenta line and a blue point.
 *  This program illustrates the two main types of methods in standard
 *  drawing—methods that draw geometric shapes and methods that
 *  control drawing parameters.
 *  The methods {@code StdDraw.line()} and {@code StdDraw.point()}
 *  draw lines and points; the methods {@code StdDraw.setPenRadius()}
 *  and {@code StdDraw.setPenColor()} control the line thickness and color.
 *  <p>
 *  <b>Points and lines.</b>
 *  You can draw points and line segments with the following methods:
 *  <ul>
 *  <li> {@link #point(double x, double y)}
 *  <li> {@link #line(double x1, double y1, double x2, double y2)}
 *  </ul>
 *  <p>
 *  The <em>x</em>- and <em>y</em>-coordinates must be in the drawing area
 *  (between 0 and 1 and by default) or the points and lines will not be visible.
 *  <p>
 *  <b>Squares, circles, rectangles, and ellipses.</b>
 *  You can draw squares, circles, rectangles, and ellipses using
 *  the following methods:
```

```
*  <ul>
*  <li> {@link #circle(double x, double y, double radius)}
*  <li> {@link #ellipse(double x, double y, double semiMajorAxis, double semiMinorAxis)}
*  <li> {@link #square(double x, double y, double halfLength)}
*  <li> {@link #rectangle(double x, double y, double halfWidth, double halfHeight)}
*  </ul>
*  <p>
*  All of these methods take as arguments the location and size of the shape.
*  The location is always specified by the <em>x</em>- and <em>y</em>-coordinates
*  of its <em>center</em>.
*  The size of a circle is specified by its radius and the size of an ellipse is
*  specified by the lengths of its semi-major and semi-minor axes.
*  The size of a square or rectangle is specified by its half-width or half-height.
*  The convention for drawing squares and rectangles is parallel to those for
*  drawing circles and ellipses, but may be unexpected to the uninitiated.
*  <p>
*  The methods above trace outlines of the given shapes. The following methods
*  draw filled versions:
*  <ul>
*  <li> {@link #filledCircle(double x, double y, double radius)}
*  <li> {@link #filledEllipse(double x, double y, double semiMajorAxis, double semiMinorAxis)}
*  <li> {@link #filledSquare(double x, double y, double radius)}
*  <li> {@link #filledRectangle(double x, double y, double halfWidth, double halfHeight)}
*  </ul>
*  <p>
*  <b>Circular arcs.</b>
*  You can draw circular arcs with the following method:
*  <ul>
*  <li> {@link #arc(double x, double y, double radius, double angle1, double angle2)}
*  </ul>
*  <p>
*  The arc is from the circle centered at (<em>x</em>, <em>y</em>) of the specified radius.
*  The arc extends from angle1 to angle2. By convention, the angles are
*  <em>polar</em> (counterclockwise angle from the <em>x</em>-axis)
*  and represented in degrees. For example, {@code StdDraw.arc(0.0, 0.0, 1.0, 0, 90)}
*  draws the arc of the unit circle from 3 o'clock (0 degrees) to 12 o'clock (90 degrees).
*  <p>
*  <b>Polygons.</b>
*  You can draw polygons with the following methods:
*  <ul>
*  <li> {@link #polygon(double[] x, double[] y)}
*  <li> {@link #filledPolygon(double[] x, double[] y)}
*  </ul>
*  <p>
*  The points in the polygon are ({@code x[i]}, {@code y[i]}).
*  For example, the following code fragment draws a filled diamond
*  with vertices (0.1, 0.2), (0.2, 0.3), (0.3, 0.2), and (0.2, 0.1):
*  <pre>
*   double[] x = { 0.1, 0.2, 0.3, 0.2 };
*   double[] y = { 0.2, 0.3, 0.2, 0.1 };
*   StdDraw.filledPolygon(x, y);
*  </pre>
*  <p>
*  <b>Pen size.</b>
*  The pen is circular, so that when you set the pen radius to <em>r</em>
*  and draw a point, you get a circle of radius <em>r</em>. Also, lines are
*  of thickness 2<em>r</em> and have rounded ends. The default pen radius
*  is 0.005 and is not affected by coordinate scaling. This default pen
*  radius is about 1/200 the width of the default canvas, so that if
*  you draw 100 points equally spaced along a horizontal or vertical line,
*  you will be able to see individual circles, but if you draw 200 such
*  points, the result will look like a line.
*  <ul>
*  <li> {@link #setPenRadius(double radius)}
*  </ul>
```

```
 *  <p>
 *  For example, {@code StdDraw.setPenRadius(0.025)} makes
 *  the thickness of the lines and the size of the points to be five times
 *  the 0.005 default.
 *  To draw points with the minimum possible radius (one pixel on typical
 *  displays), set the pen radius to 0.0.
 *  <p>
 *  <b>Pen color.</b>
 *  All geometric shapes (such as points, lines, and circles) are drawn using
 *  the current pen color. By default, it is black.
 *  You can change the pen color with the following methods:
 *  <ul>
 *  <li> {@link #setPenColor(int red, int green, int blue)}
 *  <li> {@link #setPenColor(Color color)}
 *  </ul>
 *  <p>
 *  The first method allows you to specify colors using the RGB color system.
 *  This <a href = "http://johndyer.name/lab/colorpicker/">color picker</a>
 *  is a convenient way to find a desired color.
 *  The second method allows you to specify colors using the
 *  {@link Color} data type that is discussed in Chapter 3. Until then,
 *  you can use this method with one of these predefined colors in standard drawing:
 *  {@link #BLACK}, {@link #BLUE}, {@link #CYAN}, {@link #DARK_GRAY}, {@link #GRAY},
 *  {@link #GREEN}, {@link #LIGHT_GRAY}, {@link #MAGENTA}, {@link #ORANGE},
 *  {@link #PINK}, {@link #RED}, {@link #WHITE}, {@link #YELLOW},
 *  {@link #BOOK_BLUE}, {@link #BOOK_LIGHT_BLUE}, {@link #BOOK_RED}, and
 *  {@link #PRINCETON_ORANGE}.
 *  For example, {@code StdDraw.setPenColor(StdDraw.MAGENTA)} sets the
 *  pen color to magenta.
 *  <p>
 *  <b>Canvas size.</b>
 *  By default, all drawing takes places in a 512-by-512 canvas.
 *  The canvas does not include the window title or window border.
 *  You can change the size of the canvas with the following method:
 *  <ul>
 *  <li> {@link #setCanvasSize(int width, int height)}
 *  </ul>
 *  <p>
 *  This sets the canvas size to be <em>width</em>-by-<em>height</em> pixels.
 *  It also erases the current drawing and resets the coordinate system,
 *  pen radius, pen color, and font back to their default values.
 *  Ordinarly, this method is called once, at the very beginning of a program.
 *  For example, {@code StdDraw.setCanvasSize(800, 800)}
 *  sets the canvas size to be 800-by-800 pixels.
 *  <p>
 *  <b>Canvas scale and coordinate system.</b>
 *  By default, all drawing takes places in the unit square, with (0, 0) at
 *  lower left and (1, 1) at upper right. You can change the default
 *  coordinate system with the following methods:
 *  <ul>
 *  <li> {@link #setXscale(double xmin, double xmax)}
 *  <li> {@link #setYscale(double ymin, double ymax)}
 *  <li> {@link #setScale(double min, double max)}
 *  </ul>
 *  <p>
 *  The arguments are the coordinates of the minimum and maximum
 *  <em>x</em>- or <em>y</em>-coordinates that will appear in the canvas.
 *  For example, if you  wish to use the default coordinate system but
 *  leave a small margin, you can call {@code StdDraw.setScale(-.05, 1.05)}.
 *  <p>
 *  These methods change the coordinate system for subsequent drawing
 *  commands; they do not affect previous drawings.
 *  These methods do not change the canvas size; so, if the <em>x</em>-
 *  and <em>y</em>-scales are different, squares will become rectangles
 *  and circles will become ellipses.
```

```
 *  <p>
 *  <b>Text.</b>
 *  You can use the following methods to annotate your drawings with text:
 *  <ul>
 *  <li> {@link #text(double x, double y, String text)}
 *  <li> {@link #text(double x, double y, String text, double degrees)}
 *  <li> {@link #textLeft(double x, double y, String text)}
 *  <li> {@link #textRight(double x, double y, String text)}
 *  </ul>
 *  <p>
 *  The first two methods write the specified text in the current font,
 *  centered at (<em>x</em>, <em>y</em>).
 *  The second method allows you to rotate the text.
 *  The last two methods either left- or right-align the text at (<em>x</em>, <em>y</em>).
 *  <p>
 *  The default font is a Sans Serif font with point size 16.
 *  You can use the following method to change the font:
 *  <ul>
 *  <li> {@link #setFont(Font font)}
 *  </ul>
 *  <p>
 *  You use the {@link Font} data type to specify the font. This allows you to
 *  choose the face, size, and style of the font. For example, the following
 *  code fragment sets the font to Arial Bold, 60 point.
 *  <pre>
 *   Font font = new Font("Arial", Font.BOLD, 60);
 *   StdDraw.setFont(font);
 *   StdDraw.text(0.5, 0.5, "Hello, World");
 *  </pre>
 *  <p>
 *  <b>Images.</b>
 *  You can use the following methods to add images to your drawings:
 *  <ul>
 *  <li> {@link #picture(double x, double y, String filename)}
 *  <li> {@link #picture(double x, double y, String filename, double degrees)}
 *  <li> {@link #picture(double x, double y, String filename, double scaledWidth, double
scaledHeight)}
 *  <li> {@link #picture(double x, double y, String filename, double scaledWidth, double
scaledHeight, double degrees)}
 *  </ul>
 *  <p>
 *  These methods draw the specified image, centered at (<em>x</em>, <em>y</em>).
 *  The supported image formats are JPEG, PNG, and GIF.
 *  The image will display at its native size, independent of the coordinate system.
 *  Optionally, you can rotate the image a specified number of degrees counterclockwise
 *  or rescale it to fit snugly inside a width-by-height bounding box.
 *  <p>
 *  <b>Saving to a file.</b>
 *  You save your image to a file using the <em>File → Save</em> menu option.
 *  You can also save a file programatically using the following method:
 *  <ul>
 *  <li> {@link #save(String filename)}
 *  </ul>
 *  <p>
 *  The supported image formats are JPEG and PNG. The filename must have either the
 *  extension .jpg or .png.
 *  We recommend using PNG for drawing that consist solely of geometric shapes and JPEG
 *  for drawings that contains pictures.
 *  <p>
 *  <b>Clearing the canvas.</b>
 *  To clear the entire drawing canvas, you can use the following methods:
 *  <ul>
 *  <li> {@link #clear()}
 *  <li> {@link #clear(Color color)}
 *  </ul>
```

```
*  <p>
*  The first method clears the canvas to white; the second method
*  allows you to specify a color of your choice. For example,
*  {@code StdDraw.clear(StdDraw.LIGHT_GRAY)} clears the canvas to a shade
*  of gray.
*  <p>
*  <b>Computer animations and double buffering.</b>
*  Double buffering is one of the most powerful features of standard drawing,
*  enabling computer animations.
*  The following methods control the way in which objects are drawn:
*  <ul>
*  <li> {@link #enableDoubleBuffering()}
*  <li> {@link #disableDoubleBuffering()}
*  <li> {@link #show()}
*  <li> {@link #pause(int t)}
*  </ul>
*  <p>
*  By default, double buffering is disabled, which means that as soon as you
*  call a drawing
*  method—such as {@code point()} or {@code line()}—the
*  results appear on the screen.
*  <p>
*  When double buffering is enabled by calling {@link #enableDoubleBuffering()},
*  all drawing takes place on the <em>offscreen canvas</em>. The offscreen canvas
*  is not displayed. Only when you call
*  {@link #show()} does your drawing get copied from the offscreen canvas to
*  the onscreen canvas, where it is displayed in the standard drawing window. You
*  can think of double buffering as collecting all of the lines, points, shapes,
*  and text that you tell it to draw, and then drawing them all
*  <em>simultaneously</em>, upon request.
*  <p>
*  The most important use of double buffering is to produce computer
*  animations, creating the illusion of motion by rapidly
*  displaying static drawings. To produce an animation, repeat
*  the following four steps:
*  <ul>
*  <li> Clear the offscreen canvas.
*  <li> Draw objects on the offscreen canvas.
*  <li> Copy the offscreen canvas to the onscreen canvas.
*  <li> Wait for a short while.
*  </ul>
*  <p>
*  The {@link #clear()}, {@link #show()}, and {@link #pause(int t)} methods
*  support the first, third, and fourth of these steps, respectively.
*  <p>
*  For example, this code fragment animates two balls moving in a circle.
*  <pre>
*   StdDraw.setScale(-2, +2);
*   StdDraw.enableDoubleBuffering();
*
*   for (double t = 0.0; true; t += 0.02) {
*       double x = Math.sin(t);
*       double y = Math.cos(t);
*       StdDraw.clear();
*       StdDraw.filledCircle(x, y, 0.05);
*       StdDraw.filledCircle(-x, -y, 0.05);
*       StdDraw.show();
*       StdDraw.pause(20);
*   }
*  </pre>
*  <p>
*  <b>Keyboard and mouse inputs.</b>
*  Standard drawing has very basic support for keyboard and mouse input.
*  It is much less powerful than most user interface libraries provide, but also much simpler.
*  You can use the following methods to intercept mouse events:
```

```
 *  <ul>
 *  <li> {@link #isMousePressed()}
 *  <li> {@link #mouseX()}
 *  <li> {@link #mouseY()}
 *  </ul>
 *  <p>
 *  The first method tells you whether a mouse button is currently being pressed.
 *  The last two methods tells you the <em>x</em>- and <em>y</em>-coordinates of the mouse's
 *  current position, using the same coordinate system as the canvas (the unit square, by default).
 *  You should use these methods in an animation loop that waits a short while before trying
 *  to poll the mouse for its current state.
 *  You can use the following methods to intercept keyboard events:
 *  <ul>
 *  <li> {@link #hasNextKeyTyped()}
 *  <li> {@link #nextKeyTyped()}
 *  <li> {@link #isKeyPressed(int keycode)}
 *  </ul>
 *  <p>
 *  If the user types lots of keys, they will be saved in a list until you process them.
 *  The first method tells you whether the user has typed a key (that your program has
 *  not yet processed).
 *  The second method returns the next key that the user typed (that your program has
 *  not yet processed) and removes it from the list of saved keystrokes.
 *  The third method tells you whether a key is currently being pressed.
 *  <p>
 *  <b>Accessing control parameters.</b>
 *  You can use the following methods to access the current pen color, pen radius,
 *  and font:
 *  <ul>
 *  <li> {@link #getPenColor()}
 *  <li> {@link #getPenRadius()}
 *  <li> {@link #getFont()}
 *  </ul>
 *  <p>
 *  These methods are useful when you want to temporarily change a
 *  control parameter and reset it back to its original value.
 *  <p>
 *  <b>Corner cases.</b>
 *  Here are some corner cases.
 *  <ul>
 *  <li> Drawing an object outside (or partly outside) the canvas is permitted.
 *       However, only the part of the object that appears inside the canvas
 *       will be visible.
 *  <li> Any method that is passed a {@code null} argument will throw an
 *       {@link IllegalArgumentException}.
 *  <li> Any method that is passed a {@link Double#NaN},
 *       {@link Double#POSITIVE_INFINITY}, or {@link Double#NEGATIVE_INFINITY}
 *       argument will throw an {@link IllegalArgumentException}.
 *  <li> Due to floating-point issues, an object drawn with an <em>x</em>- or
 *       <em>y</em>-coordinate that is way outside the canvas (such as the line segment
 *       from (0.5, -10^308) to (0.5, 10^308) may not be visible even in the
 *       part of the canvas where it should be.
 *  </ul>
 *  <p>
 *  <b>Performance tricks.</b>
 *  Standard drawing is capable of drawing large amounts of data.
 *  Here are a few tricks and tips:
 *  <ul>
 *  <li> Use <em>double buffering</em> for static drawing with a large
 *       number of objects.
 *       That is, call {@link #enableDoubleBuffering()} before
 *       the sequence of drawing commands and call {@link #show()} afterwards.
 *       Incrementally displaying a complex drawing while it is being
 *       created can be intolerably inefficient on many computer systems.
 *  <li> When drawing computer animations, call {@code show()}
```

```
   *       only once per frame, not after drawing each individual object.
   *  <li> If you call {@code picture()} multiple times with the same filename,
   *       Java will cache the image, so you do not incur the cost of reading
   *       from a file each time.
   *  </ul>
   *  <p>
   *  <b>Known bugs and issues.</b>
   *  <ul>
   *  <li> The {@code picture()} methods may not draw the portion of the image that is
   *       inside the canvas if the center point (<em>x</em>, <em>y</em>) is outside the
   *       canvas.
   *       This bug appears only on some systems.
   *  </ul>
   *  <p>
   *  <b>Reference.</b>
   *  For additional documentation,
   *  see <a href="https://introcs.cs.princeton.edu/15inout">Section 1.5</a> of
   *  <em>Computer Science: An Interdisciplinary Approach</em>
   *  by Robert Sedgewick and Kevin Wayne.
   *
   *  @author Robert Sedgewick
   *  @author Kevin Wayne
   */
  public final class StdDraw implements ActionListener, MouseListener, MouseMotionListener, KeyListener
  {

      /**
       *  The color black.
       */
      public static final Color BLACK = Color.BLACK;

      /**
       *  The color blue.
       */
      public static final Color BLUE = Color.BLUE;

      /**
       *  The color cyan.
       */
      public static final Color CYAN = Color.CYAN;

      /**
       *  The color dark gray.
       */
      public static final Color DARK_GRAY = Color.DARK_GRAY;

      /**
       *  The color gray.
       */
      public static final Color GRAY = Color.GRAY;

      /**
       *  The color green.
       */
      public static final Color GREEN  = Color.GREEN;

      /**
       *  The color light gray.
       */
      public static final Color LIGHT_GRAY = Color.LIGHT_GRAY;

      /**
       *  The color magenta.
       */
      public static final Color MAGENTA = Color.MAGENTA;
```

```
   /**
    *  The color orange.
    */
   public static final Color ORANGE = Color.ORANGE;

   /**
    *  The color pink.
    */
   public static final Color PINK = Color.PINK;

   /**
    *  The color red.
    */
   public static final Color RED = Color.RED;

   /**
    *  The color white.
    */
   public static final Color WHITE = Color.WHITE;

   /**
    *  The color yellow.
    */
   public static final Color YELLOW = Color.YELLOW;

   /**
    * Shade of blue used in <em>Introduction to Programming in Java</em>.
    * It is Pantone 300U. The RGB values are approximately (9, 90, 166).
    */
   public static final Color BOOK_BLUE = new Color(9, 90, 166);

   /**
    * Shade of light blue used in <em>Introduction to Programming in Java</em>.
    * The RGB values are approximately (103, 198, 243).
    */
   public static final Color BOOK_LIGHT_BLUE = new Color(103, 198, 243);

   /**
    * Shade of red used in <em>Algorithms, 4th edition</em>.
    * It is Pantone 1805U. The RGB values are approximately (150, 35, 31).
    */
   public static final Color BOOK_RED = new Color(150, 35, 31);

   /**
    * Shade of orange used in Princeton University's identity.
    * It is PMS 158. The RGB values are approximately (245, 128, 37).
    */
   public static final Color PRINCETON_ORANGE = new Color(245, 128, 37);

   // default colors
   private static final Color DEFAULT_PEN_COLOR   = BLACK;
   private static final Color DEFAULT_CLEAR_COLOR = WHITE;

   // current pen color
   private static Color penColor;

   // default canvas size is DEFAULT_SIZE-by-DEFAULT_SIZE
   private static final int DEFAULT_SIZE = 512;
   private static int width  = DEFAULT_SIZE;
   private static int height = DEFAULT_SIZE;

   // default pen radius
   private static final double DEFAULT_PEN_RADIUS = 0.002;
```

```
   // current pen radius
   private static double penRadius;

   // show we draw immediately or wait until next show?
   private static boolean defer = false;

   // boundary of drawing canvas, 0% border
   // private static final double BORDER = 0.05;
   private static final double BORDER = 0.00;
   private static final double DEFAULT_XMIN = 0.0;
   private static final double DEFAULT_XMAX = 1.0;
   private static final double DEFAULT_YMIN = 0.0;
   private static final double DEFAULT_YMAX = 1.0;
   private static double xmin, ymin, xmax, ymax;

   // for synchronization
   private static Object mouseLock = new Object();
   private static Object keyLock = new Object();

   // default font
   private static final Font DEFAULT_FONT = new Font("SansSerif", Font.PLAIN, 16);

   // current font
   private static Font font;

   // double buffered graphics
   private static BufferedImage offscreenImage, onscreenImage;
   private static Graphics2D offscreen, onscreen;

   // singleton for callbacks: avoids generation of extra .class files
   private static StdDraw std = new StdDraw();

   // the frame for drawing to the screen
   private static JFrame frame;

   // mouse state
   private static boolean isMousePressed = false;
   private static double mouseX = 0;
   private static double mouseY = 0;

   // queue of typed key characters
   private static LinkedList<Character> keysTyped;

   // set of key codes currently pressed down
   private static TreeSet<Integer> keysDown;

   // singleton pattern: client can't instantiate
   private StdDraw() { }


   // static initializer
   static {
       init();
   }

   /**
    * Sets the canvas (drawing area) to be 512-by-512 pixels.
    * This also erases the current drawing and resets the coordinate system,
    * pen radius, pen color, and font back to their default values.
    * Ordinarly, this method is called once, at the very beginning
    * of a program.
    */
   public static void setCanvasSize() {
       setCanvasSize(DEFAULT_SIZE, DEFAULT_SIZE);
   }
```

```
    /**
     * Sets the canvas (drawing area) to be <em>width</em>-by-<em>height</em> pixels.
     * This also erases the current drawing and resets the coordinate system,
     * pen radius, pen color, and font back to their default values.
     * Ordinarly, this method is called once, at the very beginning
     * of a program.
     *
     * @param  canvasWidth the width as a number of pixels
     * @param  canvasHeight the height as a number of pixels
     * @throws IllegalArgumentException unless both {@code canvasWidth} and
     *         {@code canvasHeight} are positive
     */
    public static void setCanvasSize(int canvasWidth, int canvasHeight) {
        if (canvasWidth <= 0) throw new IllegalArgumentException("width must be positive");
        if (canvasHeight <= 0) throw new IllegalArgumentException("height must be positive");
        width = canvasWidth;
        height = canvasHeight;
        init();
    }

    // init
    private static void init() {
        if (frame != null) frame.setVisible(false);
        frame = new JFrame();
        offscreenImage = new BufferedImage(2*width, 2*height, BufferedImage.TYPE_INT_ARGB);
        onscreenImage  = new BufferedImage(2*width, 2*height, BufferedImage.TYPE_INT_ARGB);
        offscreen = offscreenImage.createGraphics();
        onscreen  = onscreenImage.createGraphics();
        offscreen.scale(2.0, 2.0);  // since we made it 2x as big

        setXscale();
        setYscale();
        offscreen.setColor(DEFAULT_CLEAR_COLOR);
        offscreen.fillRect(0, 0, width, height);
        setPenColor();
        setPenRadius();
        setFont();
        clear();

        // initialize keystroke buffers
        keysTyped = new LinkedList<Character>();
        keysDown = new TreeSet<Integer>();

        // add antialiasing
        RenderingHints hints = new RenderingHints(RenderingHints.KEY_ANTIALIASING,
                                                  RenderingHints.VALUE_ANTIALIAS_ON);
        hints.put(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_QUALITY);
        offscreen.addRenderingHints(hints);

        // frame stuff
        RetinaImageIcon icon = new RetinaImageIcon(onscreenImage);
        JLabel draw = new JLabel(icon);

        draw.addMouseListener(std);
        draw.addMouseMotionListener(std);

        frame.setContentPane(draw);
        frame.addKeyListener(std);    // JLabel cannot get keyboard focus
        frame.setFocusTraversalKeysEnabled(false);  // allow VK_TAB with isKeyPressed()
        frame.setResizable(false);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);            // closes all windows
        // frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);      // closes only current
 window
        frame.setTitle("Standard Draw");
```

```
        frame.setJMenuBar(createMenuBar());
        frame.pack();
        frame.requestFocusInWindow();
        frame.setVisible(true);
    }

    // create the menu bar (changed to private)
    private static JMenuBar createMenuBar() {
        JMenuBar menuBar = new JMenuBar();
        JMenu menu = new JMenu("File");
        menuBar.add(menu);
        JMenuItem menuItem1 = new JMenuItem(" Save...   ");
        menuItem1.addActionListener(std);
        // Java 10+: replace getMenuShortcutKeyMask() with getMenuShortcutKeyMaskEx()
        menuItem1.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                 Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
        menu.add(menuItem1);
        return menuBar;
    }

   /*************************************************************************
    *  User and screen coordinate systems.
    *************************************************************************/

    // throw an IllegalArgumentException if x is NaN or infinite
    private static void validate(double x, String name) {
        if (Double.isNaN(x)) throw new IllegalArgumentException(name + " is NaN");
        if (Double.isInfinite(x)) throw new IllegalArgumentException(name + " is infinite");
    }

    // throw an IllegalArgumentException if s is null
    private static void validateNonnegative(double x, String name) {
        if (x < 0) throw new IllegalArgumentException(name + " negative");
    }

    // throw an IllegalArgumentException if s is null
    private static void validateNotNull(Object x, String name) {
        if (x == null) throw new IllegalArgumentException(name + " is null");
    }


    /**
     * Sets the <em>x</em>-scale to be the default (between 0.0 and 1.0).
     */
    public static void setXscale() {
        setXscale(DEFAULT_XMIN, DEFAULT_XMAX);
    }

    /**
     * Sets the <em>y</em>-scale to be the default (between 0.0 and 1.0).
     */
    public static void setYscale() {
        setYscale(DEFAULT_YMIN, DEFAULT_YMAX);
    }

    /**
     * Sets the <em>x</em>-scale and <em>y</em>-scale to be the default
     * (between 0.0 and 1.0).
     */
    public static void setScale() {
        setXscale();
        setYscale();
    }

    /**
```

```
     * Sets the <em>x</em>-scale to the specified range.
     *
     * @param  min the minimum value of the <em>x</em>-scale
     * @param  max the maximum value of the <em>x</em>-scale
     * @throws IllegalArgumentException if {@code (max == min)}
     * @throws IllegalArgumentException if either {@code min} or {@code max} is either NaN or
 infinite
     */
    public static void setXscale(double min, double max) {
        validate(min, "min");
        validate(max, "max");
        double size = max - min;
        if (size == 0.0) throw new IllegalArgumentException("the min and max are the same");
        synchronized (mouseLock) {
            xmin = min - BORDER * size;
            xmax = max + BORDER * size;
        }
    }

    /**
     * Sets the <em>y</em>-scale to the specified range.
     *
     * @param  min the minimum value of the <em>y</em>-scale
     * @param  max the maximum value of the <em>y</em>-scale
     * @throws IllegalArgumentException if {@code (max == min)}
     * @throws IllegalArgumentException if either {@code min} or {@code max} is either NaN or
 infinite
     */
    public static void setYscale(double min, double max) {
        validate(min, "min");
        validate(max, "max");
        double size = max - min;
        if (size == 0.0) throw new IllegalArgumentException("the min and max are the same");
        synchronized (mouseLock) {
            ymin = min - BORDER * size;
            ymax = max + BORDER * size;
        }
    }

    /**
     * Sets both the <em>x</em>-scale and <em>y</em>-scale to the (same) specified range.
     *
     * @param  min the minimum value of the <em>x</em>- and <em>y</em>-scales
     * @param  max the maximum value of the <em>x</em>- and <em>y</em>-scales
     * @throws IllegalArgumentException if {@code (max == min)}
     * @throws IllegalArgumentException if either {@code min} or {@code max} is either NaN or
 infinite
     */
    public static void setScale(double min, double max) {
        validate(min, "min");
        validate(max, "max");
        double size = max - min;
        if (size == 0.0) throw new IllegalArgumentException("the min and max are the same");
        synchronized (mouseLock) {
            xmin = min - BORDER * size;
            xmax = max + BORDER * size;
            ymin = min - BORDER * size;
            ymax = max + BORDER * size;
        }
    }

    // helper functions that scale from user coordinates to screen coordinates and back
    private static double  scaleX(double x) { return width  * (x - xmin) / (xmax - xmin); }
    private static double  scaleY(double y) { return height * (ymax - y) / (ymax - ymin); }
    private static double factorX(double w) { return w * width  / Math.abs(xmax - xmin);  }
```

```
    private static double factorY(double h) { return h * height / Math.abs(ymax - ymin);  }
    private static double   userX(double x) { return xmin + x * (xmax - xmin) / width;    }
    private static double   userY(double y) { return ymax - y * (ymax - ymin) / height;   }


    /**
     * Clears the screen to the default color (white).
     */
    public static void clear() {
        clear(DEFAULT_CLEAR_COLOR);
    }

    /**
     * Clears the screen to the specified color.
     *
     * @param color the color to make the background
     * @throws IllegalArgumentException if {@code color} is {@code null}
     */
    public static void clear(Color color) {
        validateNotNull(color, "color");
        offscreen.setColor(color);
        offscreen.fillRect(0, 0, width, height);
        offscreen.setColor(penColor);
        draw();
    }

    /**
     * Returns the current pen radius.
     *
     * @return the current value of the pen radius
     */
    public static double getPenRadius() {
        return penRadius;
    }

    /**
     * Sets the pen size to the default size (0.002).
     * The pen is circular, so that lines have rounded ends, and when you set the
     * pen radius and draw a point, you get a circle of the specified radius.
     * The pen radius is not affected by coordinate scaling.
     */
    public static void setPenRadius() {
        setPenRadius(DEFAULT_PEN_RADIUS);
    }

    /**
     * Sets the radius of the pen to the specified size.
     * The pen is circular, so that lines have rounded ends, and when you set the
     * pen radius and draw a point, you get a circle of the specified radius.
     * The pen radius is not affected by coordinate scaling.
     *
     * @param  radius the radius of the pen
     * @throws IllegalArgumentException if {@code radius} is negative, NaN, or infinite
     */
    public static void setPenRadius(double radius) {
        validate(radius, "pen radius");
        validateNonnegative(radius, "pen radius");

        penRadius = radius;
        float scaledPenRadius = (float) (radius * DEFAULT_SIZE);
        BasicStroke stroke = new BasicStroke(scaledPenRadius, BasicStroke.CAP_ROUND,
BasicStroke.JOIN_ROUND);
        // BasicStroke stroke = new BasicStroke(scaledPenRadius);
        offscreen.setStroke(stroke);
    }
```

```
    /**
     * Returns the current pen color.
     *
     * @return the current pen color
     */
    public static Color getPenColor() {
        return penColor;
    }

    /**
     * Sets the pen color to the default color (black).
     */
    public static void setPenColor() {
        setPenColor(DEFAULT_PEN_COLOR);
    }

    /**
     * Sets the pen color to the specified color.
     * <p>
     * The predefined pen colors are
     * {@code StdDraw.BLACK}, {@code StdDraw.BLUE}, {@code StdDraw.CYAN},
     * {@code StdDraw.DARK_GRAY}, {@code StdDraw.GRAY}, {@code StdDraw.GREEN},
     * {@code StdDraw.LIGHT_GRAY}, {@code StdDraw.MAGENTA}, {@code StdDraw.ORANGE},
     * {@code StdDraw.PINK}, {@code StdDraw.RED}, {@code StdDraw.WHITE}, and
     * {@code StdDraw.YELLOW}.
     *
     * @param color the color to make the pen
     * @throws IllegalArgumentException if {@code color} is {@code null}
     */
    public static void setPenColor(Color color) {
        validateNotNull(color, "color");
        penColor = color;
        offscreen.setColor(penColor);
    }

    /**
     * Sets the pen color to the specified RGB color.
     *
     * @param  red the amount of red (between 0 and 255)
     * @param  green the amount of green (between 0 and 255)
     * @param  blue the amount of blue (between 0 and 255)
     * @throws IllegalArgumentException if {@code red}, {@code green},
     *         or {@code blue} is outside its prescribed range
     */
    public static void setPenColor(int red, int green, int blue) {
        if (red   < 0 || red   >= 256) throw new IllegalArgumentException("red must be between 0 and
255");
        if (green < 0 || green >= 256) throw new IllegalArgumentException("green must be between 0
and 255");
        if (blue  < 0 || blue  >= 256) throw new IllegalArgumentException("blue must be between 0 and
255");
        setPenColor(new Color(red, green, blue));
    }

    /**
     * Returns the current font.
     *
     * @return the current font
     */
    public static Font getFont() {
        return font;
    }

    /**
```

```
     * Sets the font to the default font (sans serif, 16 point).
     */
    public static void setFont() {
        setFont(DEFAULT_FONT);
    }

    /**
     * Sets the font to the specified value.
     *
     * @param font the font
     * @throws IllegalArgumentException if {@code font} is {@code null}
     */
    public static void setFont(Font font) {
        validateNotNull(font, "font");
        StdDraw.font = font;
    }


   /***************************************************************************
    *  Drawing geometric shapes.
    ***************************************************************************/

    /**
     * Draws a line segment between (<em>x</em><sub>0</sub>, <em>y</em><sub>0</sub>) and
     * (<em>x</em><sub>1</sub>, <em>y</em><sub>1</sub>).
     *
     * @param  x0 the <em>x</em>-coordinate of one endpoint
     * @param  y0 the <em>y</em>-coordinate of one endpoint
     * @param  x1 the <em>x</em>-coordinate of the other endpoint
     * @param  y1 the <em>y</em>-coordinate of the other endpoint
     * @throws IllegalArgumentException if any coordinate is either NaN or infinite
     */
    public static void line(double x0, double y0, double x1, double y1) {
        validate(x0, "x0");
        validate(y0, "y0");
        validate(x1, "x1");
        validate(y1, "y1");
        offscreen.draw(new Line2D.Double(scaleX(x0), scaleY(y0), scaleX(x1), scaleY(y1)));
        draw();
    }

    /**
     * Draws one pixel at (<em>x</em>, <em>y</em>).
     * This method is private because pixels depend on the display.
     * To achieve the same effect, set the pen radius to 0 and call {@code point()}.
     *
     * @param  x the <em>x</em>-coordinate of the pixel
     * @param  y the <em>y</em>-coordinate of the pixel
     * @throws IllegalArgumentException if {@code x} or {@code y} is either NaN or infinite
     */
    private static void pixel(double x, double y) {
        validate(x, "x");
        validate(y, "y");
        offscreen.fillRect((int) Math.round(scaleX(x)), (int) Math.round(scaleY(y)), 1, 1);
    }

    /**
     * Draws a point centered at (<em>x</em>, <em>y</em>).
     * The point is a filled circle whose radius is equal to the pen radius.
     * To draw a single-pixel point, first set the pen radius to 0.
     *
     * @param x the <em>x</em>-coordinate of the point
     * @param y the <em>y</em>-coordinate of the point
     * @throws IllegalArgumentException if either {@code x} or {@code y} is either NaN or infinite
     */
```

```
    public static void point(double x, double y) {
        validate(x, "x");
        validate(y, "y");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double r = penRadius;
        float scaledPenRadius = (float) (r * DEFAULT_SIZE);

        // double ws = factorX(2*r);
        // double hs = factorY(2*r);
        // if (ws <= 1 && hs <= 1) pixel(x, y);
        if (scaledPenRadius <= 1) pixel(x, y);
        else offscreen.fill(new Ellipse2D.Double(xs - scaledPenRadius/2, ys - scaledPenRadius/2,
                                                 scaledPenRadius, scaledPenRadius));
        draw();
    }

    /**
     * Draws a circle of the specified radius, centered at (<em>x</em>, <em>y</em>).
     *
     * @param  x the <em>x</em>-coordinate of the center of the circle
     * @param  y the <em>y</em>-coordinate of the center of the circle
     * @param  radius the radius of the circle
     * @throws IllegalArgumentException if {@code radius} is negative
     * @throws IllegalArgumentException if any argument is either NaN or infinite
     */
    public static void circle(double x, double y, double radius) {
        validate(x, "x");
        validate(y, "y");
        validate(radius, "radius");
        validateNonnegative(radius, "radius");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2*radius);
        double hs = factorY(2*radius);
        if (ws <= 1 && hs <= 1) pixel(x, y);
        else offscreen.draw(new Ellipse2D.Double(xs - ws/2, ys - hs/2, ws, hs));
        draw();
    }

    /**
     * Draws a filled circle of the specified radius, centered at (<em>x</em>, <em>y</em>).
     *
     * @param  x the <em>x</em>-coordinate of the center of the circle
     * @param  y the <em>y</em>-coordinate of the center of the circle
     * @param  radius the radius of the circle
     * @throws IllegalArgumentException if {@code radius} is negative
     * @throws IllegalArgumentException if any argument is either NaN or infinite
     */
    public static void filledCircle(double x, double y, double radius) {
        validate(x, "x");
        validate(y, "y");
        validate(radius, "radius");
        validateNonnegative(radius, "radius");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2*radius);
        double hs = factorY(2*radius);
        if (ws <= 1 && hs <= 1) pixel(x, y);
        else offscreen.fill(new Ellipse2D.Double(xs - ws/2, ys - hs/2, ws, hs));
        draw();
    }
```

```
    /**
     * Draws an ellipse with the specified semimajor and semiminor axes,
     * centered at (<em>x</em>, <em>y</em>).
     *
     * @param  x the <em>x</em>-coordinate of the center of the ellipse
     * @param  y the <em>y</em>-coordinate of the center of the ellipse
     * @param  semiMajorAxis is the semimajor axis of the ellipse
     * @param  semiMinorAxis is the semiminor axis of the ellipse
     * @throws IllegalArgumentException if either {@code semiMajorAxis}
     *         or {@code semiMinorAxis} is negative
     * @throws IllegalArgumentException if any argument is either NaN or infinite
     */
    public static void ellipse(double x, double y, double semiMajorAxis, double semiMinorAxis) {
        validate(x, "x");
        validate(y, "y");
        validate(semiMajorAxis, "semimajor axis");
        validate(semiMinorAxis, "semiminor axis");
        validateNonnegative(semiMajorAxis, "semimajor axis");
        validateNonnegative(semiMinorAxis, "semiminor axis");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2*semiMajorAxis);
        double hs = factorY(2*semiMinorAxis);
        if (ws <= 1 && hs <= 1) pixel(x, y);
        else offscreen.draw(new Ellipse2D.Double(xs - ws/2, ys - hs/2, ws, hs));
        draw();
    }

    /**
     * Draws a filled ellipse with the specified semimajor and semiminor axes,
     * centered at (<em>x</em>, <em>y</em>).
     *
     * @param  x the <em>x</em>-coordinate of the center of the ellipse
     * @param  y the <em>y</em>-coordinate of the center of the ellipse
     * @param  semiMajorAxis is the semimajor axis of the ellipse
     * @param  semiMinorAxis is the semiminor axis of the ellipse
     * @throws IllegalArgumentException if either {@code semiMajorAxis}
     *         or {@code semiMinorAxis} is negative
     * @throws IllegalArgumentException if any argument is either NaN or infinite
     */
    public static void filledEllipse(double x, double y, double semiMajorAxis, double semiMinorAxis)
{
        validate(x, "x");
        validate(y, "y");
        validate(semiMajorAxis, "semimajor axis");
        validate(semiMinorAxis, "semiminor axis");
        validateNonnegative(semiMajorAxis, "semimajor axis");
        validateNonnegative(semiMinorAxis, "semiminor axis");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2*semiMajorAxis);
        double hs = factorY(2*semiMinorAxis);
        if (ws <= 1 && hs <= 1) pixel(x, y);
        else offscreen.fill(new Ellipse2D.Double(xs - ws/2, ys - hs/2, ws, hs));
        draw();
    }


    /**
     * Draws a circular arc of the specified radius,
     * centered at (<em>x</em>, <em>y</em>), from angle1 to angle2 (in degrees).
```

```
     *
     * @param  x the <em>x</em>-coordinate of the center of the circle
     * @param  y the <em>y</em>-coordinate of the center of the circle
     * @param  radius the radius of the circle
     * @param  angle1 the starting angle. 0 would mean an arc beginning at 3 o'clock.
     * @param  angle2 the angle at the end of the arc. For example, if
     *         you want a 90 degree arc, then angle2 should be angle1 + 90.
     * @throws IllegalArgumentException if {@code radius} is negative
     * @throws IllegalArgumentException if any argument is either NaN or infinite
     */
    public static void arc(double x, double y, double radius, double angle1, double angle2) {
        validate(x, "x");
        validate(y, "y");
        validate(radius, "arc radius");
        validate(angle1, "angle1");
        validate(angle2, "angle2");
        validateNonnegative(radius, "arc radius");

        while (angle2 < angle1) angle2 += 360;
        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2*radius);
        double hs = factorY(2*radius);
        if (ws <= 1 && hs <= 1) pixel(x, y);
        else offscreen.draw(new Arc2D.Double(xs - ws/2, ys - hs/2, ws, hs, angle1, angle2 - angle1,
Arc2D.OPEN));
        draw();
    }

    /**
     * Draws a square of the specified size, centered at (<em>x</em>, <em>y</em>).
     *
     * @param  x the <em>x</em>-coordinate of the center of the square
     * @param  y the <em>y</em>-coordinate of the center of the square
     * @param  halfLength one half the length of any side of the square
     * @throws IllegalArgumentException if {@code halfLength} is negative
     * @throws IllegalArgumentException if any argument is either NaN or infinite
     */
    public static void square(double x, double y, double halfLength) {
        validate(x, "x");
        validate(y, "y");
        validate(halfLength, "halfLength");
        validateNonnegative(halfLength, "half length");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2*halfLength);
        double hs = factorY(2*halfLength);
        if (ws <= 1 && hs <= 1) pixel(x, y);
        else offscreen.draw(new Rectangle2D.Double(xs - ws/2, ys - hs/2, ws, hs));
        draw();
    }

    /**
     * Draws a filled square of the specified size, centered at (<em>x</em>, <em>y</em>).
     *
     * @param  x the <em>x</em>-coordinate of the center of the square
     * @param  y the <em>y</em>-coordinate of the center of the square
     * @param  halfLength one half the length of any side of the square
     * @throws IllegalArgumentException if {@code halfLength} is negative
     * @throws IllegalArgumentException if any argument is either NaN or infinite
     */
    public static void filledSquare(double x, double y, double halfLength) {
        validate(x, "x");
        validate(y, "y");
```

```
        validate(halfLength, "halfLength");
        validateNonnegative(halfLength, "half length");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2*halfLength);
        double hs = factorY(2*halfLength);
        if (ws <= 1 && hs <= 1) pixel(x, y);
        else offscreen.fill(new Rectangle2D.Double(xs - ws/2, ys - hs/2, ws, hs));
        draw();
    }


    /**
     * Draws a rectangle of the specified size, centered at (<em>x</em>, <em>y</em>).
     *
     * @param  x the <em>x</em>-coordinate of the center of the rectangle
     * @param  y the <em>y</em>-coordinate of the center of the rectangle
     * @param  halfWidth one half the width of the rectangle
     * @param  halfHeight one half the height of the rectangle
     * @throws IllegalArgumentException if either {@code halfWidth} or {@code halfHeight} is negative
     * @throws IllegalArgumentException if any argument is either NaN or infinite
     */
    public static void rectangle(double x, double y, double halfWidth, double halfHeight) {
        validate(x, "x");
        validate(y, "y");
        validate(halfWidth, "halfWidth");
        validate(halfHeight, "halfHeight");
        validateNonnegative(halfWidth, "half width");
        validateNonnegative(halfHeight, "half height");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2*halfWidth);
        double hs = factorY(2*halfHeight);
        if (ws <= 1 && hs <= 1) pixel(x, y);
        else offscreen.draw(new Rectangle2D.Double(xs - ws/2, ys - hs/2, ws, hs));
        draw();
    }

    /**
     * Draws a filled rectangle of the specified size, centered at (<em>x</em>, <em>y</em>).
     *
     * @param  x the <em>x</em>-coordinate of the center of the rectangle
     * @param  y the <em>y</em>-coordinate of the center of the rectangle
     * @param  halfWidth one half the width of the rectangle
     * @param  halfHeight one half the height of the rectangle
     * @throws IllegalArgumentException if either {@code halfWidth} or {@code halfHeight} is negative
     * @throws IllegalArgumentException if any argument is either NaN or infinite
     */
    public static void filledRectangle(double x, double y, double halfWidth, double halfHeight) {
        validate(x, "x");
        validate(y, "y");
        validate(halfWidth, "halfWidth");
        validate(halfHeight, "halfHeight");
        validateNonnegative(halfWidth, "half width");
        validateNonnegative(halfHeight, "half height");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2*halfWidth);
        double hs = factorY(2*halfHeight);
        if (ws <= 1 && hs <= 1) pixel(x, y);
        else offscreen.fill(new Rectangle2D.Double(xs - ws/2, ys - hs/2, ws, hs));
        draw();
```

```
    }


    /**
     * Draws a polygon with the vertices
     * (<em>x</em><sub>0</sub>, <em>y</em><sub>0</sub>),
     * (<em>x</em><sub>1</sub>, <em>y</em><sub>1</sub>), ...,
     * (<em>x</em><sub><em>n</em>-1</sub>, <em>y</em><sub><em>n</em>-1</sub>).
     *
     * @param  x an array of all the <em>x</em>-coordinates of the polygon
     * @param  y an array of all the <em>y</em>-coordinates of the polygon
     * @throws IllegalArgumentException unless {@code x[]} and {@code y[]}
     *         are of the same length
     * @throws IllegalArgumentException if any coordinate is either NaN or infinite
     * @throws IllegalArgumentException if either {@code x[]} or {@code y[]} is {@code null}
     */
    public static void polygon(double[] x, double[] y) {
        validateNotNull(x, "x-coordinate array");
        validateNotNull(y, "y-coordinate array");
        for (int i = 0; i < x.length; i++) validate(x[i], "x[" + i + "]");
        for (int i = 0; i < y.length; i++) validate(y[i], "y[" + i + "]");

        int n1 = x.length;
        int n2 = y.length;
        if (n1 != n2) throw new IllegalArgumentException("arrays must be of the same length");
        int n = n1;
        if (n == 0) return;

        GeneralPath path = new GeneralPath();
        path.moveTo((float) scaleX(x[0]), (float) scaleY(y[0]));
        for (int i = 0; i < n; i++)
            path.lineTo((float) scaleX(x[i]), (float) scaleY(y[i]));
        path.closePath();
        offscreen.draw(path);
        draw();
    }

    /**
     * Draws a filled polygon with the vertices
     * (<em>x</em><sub>0</sub>, <em>y</em><sub>0</sub>),
     * (<em>x</em><sub>1</sub>, <em>y</em><sub>1</sub>), ...,
     * (<em>x</em><sub><em>n</em>-1</sub>, <em>y</em><sub><em>n</em>-1</sub>).
     *
     * @param  x an array of all the <em>x</em>-coordinates of the polygon
     * @param  y an array of all the <em>y</em>-coordinates of the polygon
     * @throws IllegalArgumentException unless {@code x[]} and {@code y[]}
     *         are of the same length
     * @throws IllegalArgumentException if any coordinate is either NaN or infinite
     * @throws IllegalArgumentException if either {@code x[]} or {@code y[]} is {@code null}
     */
    public static void filledPolygon(double[] x, double[] y) {
        validateNotNull(x, "x-coordinate array");
        validateNotNull(y, "y-coordinate array");
        for (int i = 0; i < x.length; i++) validate(x[i], "x[" + i + "]");
        for (int i = 0; i < y.length; i++) validate(y[i], "y[" + i + "]");

        int n1 = x.length;
        int n2 = y.length;
        if (n1 != n2) throw new IllegalArgumentException("arrays must be of the same length");
        int n = n1;
        if (n == 0) return;

        GeneralPath path = new GeneralPath();
        path.moveTo((float) scaleX(x[0]), (float) scaleY(y[0]));
        for (int i = 0; i < n; i++)
```

```
            path.lineTo((float) scaleX(x[i]), (float) scaleY(y[i]));
        path.closePath();
        offscreen.fill(path);
        draw();
    }


   /*************************************************************************
    *  Drawing images.
    *************************************************************************/
    // get an image from the given filename
    private static Image getImage(String filename) {
        if (filename == null) throw new IllegalArgumentException();

        // to read from file
        ImageIcon icon = new ImageIcon(filename);

        // try to read from URL
        if ((icon == null) || (icon.getImageLoadStatus() != MediaTracker.COMPLETE)) {
            try {
                URL url = new URL(filename);
                icon = new ImageIcon(url);
            }
            catch (MalformedURLException e) {
                /* not a url */
            }
        }

        // in case file is inside a .jar (classpath relative to StdDraw)
        if ((icon == null) || (icon.getImageLoadStatus() != MediaTracker.COMPLETE)) {
            URL url = StdDraw.class.getResource(filename);
            if (url != null)
                icon = new ImageIcon(url);
        }

        // in case file is inside a .jar (classpath relative to root of jar)
        if ((icon == null) || (icon.getImageLoadStatus() != MediaTracker.COMPLETE)) {
            URL url = StdDraw.class.getResource("/" + filename);
            if (url == null) throw new IllegalArgumentException("image " + filename + " not found");
            icon = new ImageIcon(url);
        }

        return icon.getImage();
    }

   /*************************************************************************
    * [Summer 2016] Should we update to use ImageIO instead of ImageIcon()?
    *               Seems to have some issues loading images on some systems
    *               and slows things down on other systems.
    *               especially if you don't call ImageIO.setUseCache(false)
    *               One advantage is that it returns a BufferedImage.
    *************************************************************************/
/*
    private static BufferedImage getImage(String filename) {
        if (filename == null) throw new IllegalArgumentException();

        // from a file or URL
        try {
            URL url = new URL(filename);
            BufferedImage image = ImageIO.read(url);
            return image;
        }
        catch (IOException e) {
            // ignore
        }
```

```
        // in case file is inside a .jar (classpath relative to StdDraw)
        try {
            URL url = StdDraw.class.getResource(filename);
            BufferedImage image = ImageIO.read(url);
            return image;
        }
        catch (IOException e) {
            // ignore
        }

        // in case file is inside a .jar (classpath relative to root of jar)
        try {
            URL url = StdDraw.class.getResource("/" + filename);
            BufferedImage image = ImageIO.read(url);
            return image;
        }
        catch (IOException e) {
            // ignore
        }
        throw new IllegalArgumentException("image " + filename + " not found");
    }
*/
    /**
     * Draws the specified image centered at (<em>x</em>, <em>y</em>).
     * The supported image formats are JPEG, PNG, and GIF.
     * As an optimization, the picture is cached, so there is no performance
     * penalty for redrawing the same image multiple times (e.g., in an animation).
     * However, if you change the picture file after drawing it, subsequent
     * calls will draw the original picture.
     *
     * @param  x the center <em>x</em>-coordinate of the image
     * @param  y the center <em>y</em>-coordinate of the image
     * @param  filename the name of the image/picture, e.g., "ball.gif"
     * @throws IllegalArgumentException if the image filename is invalid
     * @throws IllegalArgumentException if either {@code x} or {@code y} is either NaN or infinite
     */
    public static void picture(double x, double y, String filename) {
        validate(x, "x");
        validate(y, "y");
        validateNotNull(filename, "filename");

        // BufferedImage image = getImage(filename);
        Image image = getImage(filename);
        double xs = scaleX(x);
        double ys = scaleY(y);
        // int ws = image.getWidth();    // can call only if image is a BufferedImage
        // int hs = image.getHeight();
        int ws = image.getWidth(null);
        int hs = image.getHeight(null);
        if (ws < 0 || hs < 0) throw new IllegalArgumentException("image " + filename + " is
corrupt");

        offscreen.drawImage(image, (int) Math.round(xs - ws/2.0), (int) Math.round(ys - hs/2.0),
null);
        draw();
    }

    /**
     * Draws the specified image centered at (<em>x</em>, <em>y</em>),
     * rotated given number of degrees.
     * The supported image formats are JPEG, PNG, and GIF.
     *
     * @param  x the center <em>x</em>-coordinate of the image
     * @param  y the center <em>y</em>-coordinate of the image
```

```
     * @param  filename the name of the image/picture, e.g., "ball.gif"
     * @param  degrees is the number of degrees to rotate counterclockwise
     * @throws IllegalArgumentException if the image filename is invalid
     * @throws IllegalArgumentException if {@code x}, {@code y}, {@code degrees} is NaN or infinite
     * @throws IllegalArgumentException if {@code filename} is {@code null}
     */
    public static void picture(double x, double y, String filename, double degrees) {
        validate(x, "x");
        validate(y, "y");
        validate(degrees, "degrees");
        validateNotNull(filename, "filename");

        // BufferedImage image = getImage(filename);
        Image image = getImage(filename);
        double xs = scaleX(x);
        double ys = scaleY(y);
        // int ws = image.getWidth();    // can call only if image is a BufferedImage
        // int hs = image.getHeight();
        int ws = image.getWidth(null);
        int hs = image.getHeight(null);
        if (ws < 0 || hs < 0) throw new IllegalArgumentException("image " + filename + " is
corrupt");

        offscreen.rotate(Math.toRadians(-degrees), xs, ys);
        offscreen.drawImage(image, (int) Math.round(xs - ws/2.0), (int) Math.round(ys - hs/2.0),
null);
        offscreen.rotate(Math.toRadians(+degrees), xs, ys);

        draw();
    }

    /**
     * Draws the specified image centered at (<em>x</em>, <em>y</em>),
     * rescaled to the specified bounding box.
     * The supported image formats are JPEG, PNG, and GIF.
     *
     * @param  x the center <em>x</em>-coordinate of the image
     * @param  y the center <em>y</em>-coordinate of the image
     * @param  filename the name of the image/picture, e.g., "ball.gif"
     * @param  scaledWidth the width of the scaled image (in screen coordinates)
     * @param  scaledHeight the height of the scaled image (in screen coordinates)
     * @throws IllegalArgumentException if either {@code scaledWidth}
     *         or {@code scaledHeight} is negative
     * @throws IllegalArgumentException if the image filename is invalid
     * @throws IllegalArgumentException if {@code x} or {@code y} is either NaN or infinite
     * @throws IllegalArgumentException if {@code filename} is {@code null}
     */
    public static void picture(double x, double y, String filename, double scaledWidth, double
scaledHeight) {
        validate(x, "x");
        validate(y, "y");
        validate(scaledWidth, "scaled width");
        validate(scaledHeight, "scaled height");
        validateNotNull(filename, "filename");
        validateNonnegative(scaledWidth, "scaled width");
        validateNonnegative(scaledHeight, "scaled height");

        Image image = getImage(filename);
        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(scaledWidth);
        double hs = factorY(scaledHeight);
        if (ws < 0 || hs < 0) throw new IllegalArgumentException("image " + filename + " is
corrupt");
        if (ws <= 1 && hs <= 1) pixel(x, y);
```

```
        else {
            offscreen.drawImage(image, (int) Math.round(xs - ws/2.0),
                                       (int) Math.round(ys - hs/2.0),
                                       (int) Math.round(ws),
                                       (int) Math.round(hs), null);
        }
        draw();
    }


    /**
     * Draws the specified image centered at (<em>x</em>, <em>y</em>), rotated
     * given number of degrees, and rescaled to the specified bounding box.
     * The supported image formats are JPEG, PNG, and GIF.
     *
     * @param  x the center <em>x</em>-coordinate of the image
     * @param  y the center <em>y</em>-coordinate of the image
     * @param  filename the name of the image/picture, e.g., "ball.gif"
     * @param  scaledWidth the width of the scaled image (in screen coordinates)
     * @param  scaledHeight the height of the scaled image (in screen coordinates)
     * @param  degrees is the number of degrees to rotate counterclockwise
     * @throws IllegalArgumentException if either {@code scaledWidth}
     *         or {@code scaledHeight} is negative
     * @throws IllegalArgumentException if the image filename is invalid
     */
    public static void picture(double x, double y, String filename, double scaledWidth, double
 scaledHeight, double degrees) {
        validate(x, "x");
        validate(y, "y");
        validate(scaledWidth, "scaled width");
        validate(scaledHeight, "scaled height");
        validate(degrees, "degrees");
        validateNotNull(filename, "filename");
        validateNonnegative(scaledWidth, "scaled width");
        validateNonnegative(scaledHeight, "scaled height");

        Image image = getImage(filename);
        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(scaledWidth);
        double hs = factorY(scaledHeight);
        if (ws < 0 || hs < 0) throw new IllegalArgumentException("image " + filename + " is
 corrupt");
        if (ws <= 1 && hs <= 1) pixel(x, y);

        offscreen.rotate(Math.toRadians(-degrees), xs, ys);
        offscreen.drawImage(image, (int) Math.round(xs - ws/2.0),
                                   (int) Math.round(ys - hs/2.0),
                                   (int) Math.round(ws),
                                   (int) Math.round(hs), null);
        offscreen.rotate(Math.toRadians(+degrees), xs, ys);

        draw();
    }

   /***************************************************************************
    *  Drawing text.
    ***************************************************************************/

    /**
     * Writes the given text string in the current font, centered at (<em>x</em>, <em>y</em>).
     *
     * @param  x the center <em>x</em>-coordinate of the text
     * @param  y the center <em>y</em>-coordinate of the text
     * @param  text the text to write
```

```
     * @throws IllegalArgumentException if {@code text} is {@code null}
     * @throws IllegalArgumentException if {@code x} or {@code y} is either NaN or infinite
     */
    public static void text(double x, double y, String text) {
        validate(x, "x");
        validate(y, "y");
        validateNotNull(text, "text");

        offscreen.setFont(font);
        FontMetrics metrics = offscreen.getFontMetrics();
        double xs = scaleX(x);
        double ys = scaleY(y);
        int ws = metrics.stringWidth(text);
        int hs = metrics.getDescent();
        offscreen.drawString(text, (float) (xs - ws/2.0), (float) (ys + hs));
        draw();
    }

    /**
     * Writes the given text string in the current font, centered at (<em>x</em>, <em>y</em>) and
     * rotated by the specified number of degrees.
     * @param  x the center <em>x</em>-coordinate of the text
     * @param  y the center <em>y</em>-coordinate of the text
     * @param  text the text to write
     * @param  degrees is the number of degrees to rotate counterclockwise
     * @throws IllegalArgumentException if {@code text} is {@code null}
     * @throws IllegalArgumentException if {@code x}, {@code y}, or {@code degrees} is either NaN or
infinite
     */
    public static void text(double x, double y, String text, double degrees) {
        validate(x, "x");
        validate(y, "y");
        validate(degrees, "degrees");
        validateNotNull(text, "text");

        double xs = scaleX(x);
        double ys = scaleY(y);
        offscreen.rotate(Math.toRadians(-degrees), xs, ys);
        text(x, y, text);
        offscreen.rotate(Math.toRadians(+degrees), xs, ys);
    }


    /**
     * Writes the given text string in the current font, left-aligned at (<em>x</em>, <em>y</em>).
     * @param  x the <em>x</em>-coordinate of the text
     * @param  y the <em>y</em>-coordinate of the text
     * @param  text the text
     * @throws IllegalArgumentException if {@code text} is {@code null}
     * @throws IllegalArgumentException if {@code x} or {@code y} is either NaN or infinite
     */
    public static void textLeft(double x, double y, String text) {
        validate(x, "x");
        validate(y, "y");
        validateNotNull(text, "text");

        offscreen.setFont(font);
        FontMetrics metrics = offscreen.getFontMetrics();
        double xs = scaleX(x);
        double ys = scaleY(y);
        int hs = metrics.getDescent();
        offscreen.drawString(text, (float) xs, (float) (ys + hs));
        draw();
    }
```

```
    /**
     * Writes the given text string in the current font, right-aligned at (<em>x</em>, <em>y</em>).
     *
     * @param  x the <em>x</em>-coordinate of the text
     * @param  y the <em>y</em>-coordinate of the text
     * @param  text the text to write
     * @throws IllegalArgumentException if {@code text} is {@code null}
     * @throws IllegalArgumentException if {@code x} or {@code y} is either NaN or infinite
     */
    public static void textRight(double x, double y, String text) {
        validate(x, "x");
        validate(y, "y");
        validateNotNull(text, "text");

        offscreen.setFont(font);
        FontMetrics metrics = offscreen.getFontMetrics();
        double xs = scaleX(x);
        double ys = scaleY(y);
        int ws = metrics.stringWidth(text);
        int hs = metrics.getDescent();
        offscreen.drawString(text, (float) (xs - ws), (float) (ys + hs));
        draw();
    }


    /**
     * Copies the offscreen buffer to the onscreen buffer, pauses for t milliseconds
     * and enables double buffering.
     * @param t number of milliseconds
     * @deprecated replaced by {@link #enableDoubleBuffering()}, {@link #show()}, and {@link
#pause(int t)}
     */
    @Deprecated
    public static void show(int t) {
        validateNonnegative(t, "t");
        show();
        pause(t);
        enableDoubleBuffering();
    }

    /**
     * Pauses for t milliseconds. This method is intended to support computer animations.
     * @param t number of milliseconds
     */
    public static void pause(int t) {
        validateNonnegative(t, "t");
        try {
            Thread.sleep(t);
        }
        catch (InterruptedException e) {
            System.out.println("Error sleeping");
        }
    }

    /**
     * Copies offscreen buffer to onscreen buffer. There is no reason to call
     * this method unless double buffering is enabled.
     */
    public static void show() {
        onscreen.drawImage(offscreenImage, 0, 0, null);
        frame.repaint();
    }

    // draw onscreen if defer is false
    private static void draw() {
```

```
        if (!defer) show();
    }

    /**
     * Enables double buffering. All subsequent calls to
     * drawing methods such as {@code line()}, {@code circle()},
     * and {@code square()} will be deferred until the next call
     * to show(). Useful for animations.
     */
    public static void enableDoubleBuffering() {
        defer = true;
    }

    /**
     * Disables double buffering. All subsequent calls to
     * drawing methods such as {@code line()}, {@code circle()},
     * and {@code square()} will be displayed on screen when called.
     * This is the default.
     */
    public static void disableDoubleBuffering() {
        defer = false;
    }


   /*************************************************************************
    *  Save drawing to a file.
    *************************************************************************/

    /**
     * Saves the drawing to using the specified filename.
     * The supported image formats are JPEG and PNG;
     * the filename suffix must be {@code .jpg} or {@code .png}.
     *
     * @param  filename the name of the file with one of the required suffixes
     * @throws IllegalArgumentException if {@code filename} is {@code null}
     */
    public static void save(String filename) {
        validateNotNull(filename, "filename");
        File file = new File(filename);
        String suffix = filename.substring(filename.lastIndexOf('.') + 1);

        // png files
        if ("png".equalsIgnoreCase(suffix)) {
            try {
                ImageIO.write(onscreenImage, suffix, file);
            }
            catch (IOException e) {
                e.printStackTrace();
            }
        }

        // need to change from ARGB to RGB for JPEG
        // reference: http://archives.java.sun.com/cgi-bin/wa?A2=ind0404&L=java2d-interest&D=0&P=2727
        else if ("jpg".equalsIgnoreCase(suffix)) {
            WritableRaster raster = onscreenImage.getRaster();
            WritableRaster newRaster;
            newRaster = raster.createWritableChild(0, 0, width, height, 0, 0, new int[] {0, 1, 2});
            DirectColorModel cm = (DirectColorModel) onscreenImage.getColorModel();
            DirectColorModel newCM = new DirectColorModel(cm.getPixelSize(),
                                                          cm.getRedMask(),
                                                          cm.getGreenMask(),
                                                          cm.getBlueMask());
            BufferedImage rgbBuffer = new BufferedImage(newCM, newRaster, false,  null);
            try {
                ImageIO.write(rgbBuffer, suffix, file);
```

```
            }
            catch (IOException e) {
                e.printStackTrace();
            }
        }

        else {
            System.out.println("Invalid image file type: " + suffix);
        }
    }


    /**
     * This method cannot be called directly.
     */
    @Override
    public void actionPerformed(ActionEvent e) {
        FileDialog chooser = new FileDialog(StdDraw.frame, "Use a .png or .jpg extension",
 FileDialog.SAVE);
        chooser.setVisible(true);
        String filename = chooser.getFile();
        if (filename != null) {
            StdDraw.save(chooser.getDirectory() + File.separator + chooser.getFile());
        }
    }


   /*************************************************************************
    *  Mouse interactions.
    *************************************************************************/

    /**
     * Returns true if the mouse is being pressed.
     *
     * @return {@code true} if the mouse is being pressed; {@code false} otherwise
     */
    public static boolean isMousePressed() {
        synchronized (mouseLock) {
            return isMousePressed;
        }
    }

    /**
     * Returns true if the mouse is being pressed.
     *
     * @return {@code true} if the mouse is being pressed; {@code false} otherwise
     * @deprecated replaced by {@link #isMousePressed()}
     */
    @Deprecated
    public static boolean mousePressed() {
        synchronized (mouseLock) {
            return isMousePressed;
        }
    }

    /**
     * Returns the <em>x</em>-coordinate of the mouse.
     *
     * @return the <em>x</em>-coordinate of the mouse
     */
    public static double mouseX() {
        synchronized (mouseLock) {
            return mouseX;
        }
    }
```

```
    /**
     * Returns the <em>y</em>-coordinate of the mouse.
     *
     * @return <em>y</em>-coordinate of the mouse
     */
    public static double mouseY() {
        synchronized (mouseLock) {
            return mouseY;
        }
    }


    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseClicked(MouseEvent e) {
        // this body is intentionally left empty
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseEntered(MouseEvent e) {
        // this body is intentionally left empty
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseExited(MouseEvent e) {
        // this body is intentionally left empty
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mousePressed(MouseEvent e) {
        synchronized (mouseLock) {
            mouseX = StdDraw.userX(e.getX());
            mouseY = StdDraw.userY(e.getY());
            isMousePressed = true;
        }
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseReleased(MouseEvent e) {
        synchronized (mouseLock) {
            isMousePressed = false;
        }
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseDragged(MouseEvent e)  {
        synchronized (mouseLock) {
```

```
            mouseX = StdDraw.userX(e.getX());
            mouseY = StdDraw.userY(e.getY());
        }
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseMoved(MouseEvent e) {
        synchronized (mouseLock) {
            mouseX = StdDraw.userX(e.getX());
            mouseY = StdDraw.userY(e.getY());
        }
    }


   /*****************************************************************************
    *  Keyboard interactions.
    *****************************************************************************/

    /**
     * Returns true if the user has typed a key (that has not yet been processed).
     *
     * @return {@code true} if the user has typed a key (that has not yet been processed
     *         by {@link #nextKeyTyped()}; {@code false} otherwise
     */
    public static boolean hasNextKeyTyped() {
        synchronized (keyLock) {
            return !keysTyped.isEmpty();
        }
    }

    /**
     * Returns the next key that was typed by the user (that your program has not already processed).
     * This method should be preceded by a call to {@link #hasNextKeyTyped()} to ensure
     * that there is a next key to process.
     * This method returns a Unicode character corresponding to the key
     * typed (such as {@code 'a'} or {@code 'A'}).
     * It cannot identify action keys (such as F1 and arrow keys)
     * or modifier keys (such as control).
     *
     * @return the next key typed by the user (that your program has not already processed).
     * @throws NoSuchElementException if there is no remaining key
     */
    public static char nextKeyTyped() {
        synchronized (keyLock) {
            if (keysTyped.isEmpty()) {
                throw new NoSuchElementException("your program has already processed all
keystrokes");
            }
            return keysTyped.remove(keysTyped.size() - 1);
            // return keysTyped.removeLast();
        }
    }

    /**
     * Returns true if the given key is being pressed.
     * <p>
     * This method takes the keycode (corresponding to a physical key)
    *  as an argument. It can handle action keys
     * (such as F1 and arrow keys) and modifier keys (such as shift and control).
     * See {@link KeyEvent} for a description of key codes.
     *
     * @param  keycode the key to check if it is being pressed
```

```
     * @return {@code true} if {@code keycode} is currently being pressed;
     *         {@code false} otherwise
     */
    public static boolean isKeyPressed(int keycode) {
        synchronized (keyLock) {
            return keysDown.contains(keycode);
        }
    }


    /**
     * This method cannot be called directly.
     */
    @Override
    public void keyTyped(KeyEvent e) {
        synchronized (keyLock) {
            keysTyped.addFirst(e.getKeyChar());
        }
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void keyPressed(KeyEvent e) {
        synchronized (keyLock) {
            keysDown.add(e.getKeyCode());
        }
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void keyReleased(KeyEvent e) {
        synchronized (keyLock) {
            keysDown.remove(e.getKeyCode());
        }
    }


   /***************************************************************************
    *  For improved resolution on Mac Retina displays.
    ***************************************************************************/

    private static class RetinaImageIcon extends ImageIcon {

        public RetinaImageIcon(Image image) {
            super(image);
        }

        public int getIconWidth() {
            return super.getIconWidth() / 2;
        }

        /**
         * Gets the height of the icon.
         *
         * @return the height in pixels of this icon
         */
        public int getIconHeight() {
            return super.getIconHeight() / 2;
        }

        public synchronized void paintIcon(Component c, Graphics g, int x, int y) {
```

```
            Graphics2D g2 = (Graphics2D) g.create();

g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,RenderingHints.VALUE_INTERPOLATION_BICUBIC);
            g2.setRenderingHint(RenderingHints.KEY_RENDERING,RenderingHints.VALUE_RENDER_QUALITY);
            g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,RenderingHints.VALUE_ANTIALIAS_ON);
            g2.scale(0.5, 0.5);
            super.paintIcon(c, g2, x * 2, y * 2);
            g2.dispose();
        }
    }


    /**
     * Test client.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        StdDraw.square(0.2, 0.8, 0.1);
        StdDraw.filledSquare(0.8, 0.8, 0.2);
        StdDraw.circle(0.8, 0.2, 0.2);

        StdDraw.setPenColor(StdDraw.BOOK_RED);
        StdDraw.setPenRadius(0.02);
        StdDraw.arc(0.8, 0.2, 0.1, 200, 45);

        // draw a blue diamond
        StdDraw.setPenRadius();
        StdDraw.setPenColor(StdDraw.BOOK_BLUE);
        double[] x = { 0.1, 0.2, 0.3, 0.2 };
        double[] y = { 0.2, 0.3, 0.2, 0.1 };
        StdDraw.filledPolygon(x, y);

        // text
        StdDraw.setPenColor(StdDraw.BLACK);
        StdDraw.text(0.2, 0.5, "black text");
        StdDraw.setPenColor(StdDraw.WHITE);
        StdDraw.text(0.8, 0.8, "white text");
    }

}
```