Class StdDraw

Object
    StdDraw

All Implemented Interfaces:

ActionListener, KeyListener, MouseListener, MouseMotionListener, EventListener

---

```
public final class StdDraw
extends Object
implements ActionListener, MouseListener, MouseMotionListener, KeyListener
```

The `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.

Getting started. To use this class, you must have `StdDraw.class` in your Java classpath. If you used our autoinstaller, you should be all set. Otherwise, either download stdlib.jar and add to your Java classpath or download StdDraw.java and put a copy in your working directory.

Now, type the following short program into your editor:

```
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);
    }
}
```

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 `StdDraw.line()` and `StdDraw.point()` draw lines and points; the methods `StdDraw.setPenRadius()` and `StdDraw.setPenColor()` control the line thickness and color.

Points and lines. You can draw points and line segments with the following methods:

- `point(double x, double y)`
- `line(double x1, double y1, double x2, double y2)`

The *x*- and *y*-coordinates must be in the drawing area (between 0 and 1 and by default) or the points and lines will not be visible.

Squares, circles, rectangles, and ellipses. You can draw squares, circles, rectangles, and ellipses using the following methods:

- `circle(double x, double y, double radius)`
- `ellipse(double x, double y, double semiMajorAxis, double semiMinorAxis)`
- `square(double x, double y, double halfLength)`
- `rectangle(double x, double y, double halfWidth, double halfHeight)`

All of these methods take as arguments the location and size of the shape. The location is always specified by the *x*- and *y*-coordinates of its *center*. 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.

The methods above trace outlines of the given shapes. The following methods draw filled versions:

- `filledCircle(double x, double y, double radius)`
- `filledEllipse(double x, double y, double semiMajorAxis, double semiMinorAxis)`
- `filledSquare(double x, double y, double radius)`
- `filledRectangle(double x, double y, double halfWidth, double halfHeight)`

Circular arcs. You can draw circular arcs with the following method:

- `arc(double x, double y, double radius, double angle1, double angle2)`

The arc is from the circle centered at (*x*, *y*) of the specified radius. The arc extends from angle1 to angle2. By convention, the angles are *polar* (counterclockwise angle from the *x*-axis) and represented in degrees. For example, `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).

Polygons. You can draw polygons with the following methods:

- `polygon(double[] x, double[] y)`
- `filledPolygon(double[] x, double[] y)`

The points in the polygon are (`x[i]`, `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):

```
double[] x = { 0.1, 0.2, 0.3, 0.2 };
double[] y = { 0.2, 0.3, 0.2, 0.1 };
StdDraw.filledPolygon(x, y);
```

Pen size. The pen is circular, so that when you set the pen radius to *r* and draw a point, you get a circle of radius *r*. Also, lines are of thickness 2*r* 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.

- `setPenRadius(double radius)`

For example, `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.

Pen color. 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:

- `setPenColor(int red, int green, int blue)`
- `setPenColor(Color color)`

The first method allows you to specify colors using the RGB color system. This color picker is a convenient way to find a desired color. The second method allows you to specify colors using the `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: `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, `YELLOW`, `BOOK_BLUE`, `BOOK_LIGHT_BLUE`, `BOOK_RED`, and `PRINCETON_ORANGE`. For example, `StdDraw.setPenColor(StdDraw.MAGENTA)` sets the pen color to magenta.

Canvas size. 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:

- `setCanvasSize(int width, int height)`

This sets the canvas size to be *width*-by-*height* 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, `StdDraw.setCanvasSize(800, 800)` sets the canvas size to be 800-by-800 pixels.

Canvas scale and coordinate system. 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:

- `setXscale(double xmin, double xmax)`
- `setYscale(double ymin, double ymax)`
- `setScale(double min, double max)`

The arguments are the coordinates of the minimum and maximum *x*- or *y*-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 `StdDraw.setScale(-.05, 1.05)`.

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 *x*- and *y*-scales are different, squares will become rectangles and circles will become ellipses.

Text. You can use the following methods to annotate your drawings with text:

- `text(double x, double y, String text)`
- `text(double x, double y, String text, double degrees)`
- `textLeft(double x, double y, String text)`
- `textRight(double x, double y, String text)`

The first two methods write the specified text in the current font, centered at (*x*, *y*). The second method allows you to rotate the text. The last two methods either left- or right-align the text at (*x*, *y*).

The default font is a Sans Serif font with point size 16. You can use the following method to change the font:

- `setFont(Font font)`

You use the `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.

```
Font font = new Font("Arial", Font.BOLD, 60);
StdDraw.setFont(font);
StdDraw.text(0.5, 0.5, "Hello, World");
```

Images. You can use the following methods to add images to your drawings:

- `picture(double x, double y, String filename)`
- `picture(double x, double y, String filename, double degrees)`
- `picture(double x, double y, String filename, double scaledWidth, double scaledHeight)`
- `picture(double x, double y, String filename, double scaledWidth, double scaledHeight, double degrees)`

These methods draw the specified image, centered at (*x*, *y*). 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.

Saving to a file. You save your image to a file using the *File → Save* menu option. You can also save a file programatically using the following method:

- `save(String filename)`

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.

Clearing the canvas. To clear the entire drawing canvas, you can use the following methods:

- `clear()`
- `clear(Color color)`

The first method clears the canvas to white; the second method allows you to specify a color of your choice. For example, `StdDraw.clear(StdDraw.LIGHT_GRAY)` clears the canvas to a shade of gray.

Computer animations and double buffering. 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:

- `enableDoubleBuffering()`
- `disableDoubleBuffering()`
- `show()`
- `pause(int t)`

By default, double buffering is disabled, which means that as soon as you call a drawing method—such as `point()` or `line()`—the results appear on the screen.

When double buffering is enabled by calling `enableDoubleBuffering()`, all drawing takes place on the *offscreen canvas*. The offscreen canvas is not displayed. Only when you call `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 *simultaneously*, upon request.

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:

- Clear the offscreen canvas.
- Draw objects on the offscreen canvas.
- Copy the offscreen canvas to the onscreen canvas.
- Wait for a short while.

The `clear()`, `show()`, and `pause(int t)` methods support the first, third, and fourth of these steps, respectively.

For example, this code fragment animates two balls moving in a circle.

```
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);
}
```

Keyboard and mouse inputs. 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:

- `isMousePressed()`
- `mouseX()`
- `mouseY()`

The first method tells you whether a mouse button is currently being pressed. The last two methods tells you the *x*- and *y*-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:

- `hasNextKeyTyped()`
- `nextKeyTyped()`
- `isKeyPressed(int keycode)`

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.

Accessing control parameters. You can use the following methods to access the current pen color, pen radius, and font:

- `getPenColor()`
- `getPenRadius()`
- `getFont()`

These methods are useful when you want to temporarily change a control parameter and reset it back to its original value.

Corner cases. Here are some corner cases.

- 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.
- Any method that is passed a `null` argument will throw an `IllegalArgumentException`.
- Any method that is passed a `Double.NaN`, `Double.POSITIVE_INFINITY`, or `Double.NEGATIVE_INFINITY` argument will throw an `IllegalArgumentException`.
- Due to floating-point issues, an object drawn with an *x*- or *y*-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.

Performance tricks. Standard drawing is capable of drawing large amounts of data. Here are a few tricks and tips:

- Use *double buffering* for static drawing with a large number of objects. That is, call `enableDoubleBuffering()` before the sequence of drawing commands and call `show()` afterwards. Incrementally displaying a complex drawing while it is being created can be intolerably inefficient on many computer systems.
- When drawing computer animations, call `show()` only once per frame, not after drawing each individual object.
- If you call `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.

Known bugs and issues.

- The `picture()` methods may not draw the portion of the image that is inside the canvas if the center point (*x*, *y*) is outside the canvas. This bug appears only on some systems.

Reference. For additional documentation, see Section 1.5 of *Computer Science: An Interdisciplinary Approach* by Robert Sedgewick and Kevin Wayne.

Author:

Robert Sedgewick, Kevin Wayne

Field Summary

Fields

Modifier and Type	Field and Description
static Color	BLACK The color black.
static Color	BLUE The color blue.
static Color	BOOK_BLUE Shade of blue used in *Introduction to Programming in Java*.
static Color	BOOK_LIGHT_BLUE Shade of light blue used in *Introduction to Programming in Java*.
static Color	BOOK_RED Shade of red used in *Algorithms, 4th edition*.
static Color	CYAN The color cyan.
static Color	DARK_GRAY The color dark gray.
static Color	GRAY The color gray.
static Color	GREEN The color green.
static Color	LIGHT_GRAY The color light gray.
static Color	MAGENTA The color magenta.
static Color	ORANGE The color orange.
static Color	PINK The color pink.
static Color	PRINCETON_ORANGE Shade of orange used in Princeton University's identity.
static Color	RED The color red.
static Color	WHITE The color white.
static Color	YELLOW The color yellow.

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method and Description
void	actionPerformed(ActionEvent e) This method cannot be called directly.
static void	arc(double x, double y, double radius, double angle1, double angle2) Draws a circular arc of the specified radius, centered at (*x*, *y*), from angle1 to angle2 (in degrees).
static void	circle(double x, double y, double radius) Draws a circle of the specified radius, centered at (*x*, *y*).

static void	**clear**() Clears the screen to the default color (white).
static void	**clear**(Color color) Clears the screen to the specified color.
static void	**disableDoubleBuffering**() Disables double buffering.
static void	**ellipse**(double x, double y, double semiMajorAxis, double semiMinorAxis) Draws an ellipse with the specified semimajor and semiminor axes, centered at (*x*, *y*).
static void	**enableDoubleBuffering**() Enables double buffering.
static void	**filledCircle**(double x, double y, double radius) Draws a filled circle of the specified radius, centered at (*x*, *y*).
static void	**filledEllipse**(double x, double y, double semiMajorAxis, double semiMinorAxis) Draws a filled ellipse with the specified semimajor and semiminor axes, centered at (*x*, *y*).
static void	**filledPolygon**(double[] x, double[] y) Draws a filled polygon with the vertices (x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1}).
static void	**filledRectangle**(double x, double y, double halfWidth, double halfHeight) Draws a filled rectangle of the specified size, centered at (*x*, *y*).
static void	**filledSquare**(double x, double y, double halfLength) Draws a filled square of the specified size, centered at (*x*, *y*).
static Font	**getFont**() Returns the current font.
static Color	**getPenColor**() Returns the current pen color.
static double	**getPenRadius**() Returns the current pen radius.
static boolean	**hasNextKeyTyped**() Returns true if the user has typed a key (that has not yet been processed).
static boolean	**isKeyPressed**(int keycode) Returns true if the given key is being pressed.
static boolean	**isMousePressed**() Returns true if the mouse is being pressed.
void	**keyPressed**(KeyEvent e) This method cannot be called directly.
void	**keyReleased**(KeyEvent e) This method cannot be called directly.
void	**keyTyped**(KeyEvent e) This method cannot be called directly.
static void	**line**(double x0, double y0, double x1, double y1) Draws a line segment between (x_0, y_0) and (x_1, y_1).
static void	**main**(String[] args) Test client.
void	**mouseClicked**(MouseEvent e) This method cannot be called directly.
void	**mouseDragged**(MouseEvent e) This method cannot be called directly.
void	**mouseEntered**(MouseEvent e) This method cannot be called directly.
void	**mouseExited**(MouseEvent e) This method cannot be called directly.
void	**mouseMoved**(MouseEvent e) This method cannot be called directly.

static boolean	**mousePressed**() Deprecated. replaced by **isMousePressed()**
void	**mousePressed**(**MouseEvent** e) This method cannot be called directly.
void	**mouseReleased**(**MouseEvent** e) This method cannot be called directly.
static double	**mouseX**() Returns the *x*-coordinate of the mouse.
static double	**mouseY**() Returns the *y*-coordinate of the mouse.
static char	**nextKeyTyped**() Returns the next key that was typed by the user (that your program has not already processed).
static void	**pause**(int t) Pauses for t milliseconds.
static void	**picture**(double x, double y, **String** filename) Draws the specified image centered at (*x*, *y*).
static void	**picture**(double x, double y, **String** filename, double degrees) Draws the specified image centered at (*x*, *y*), rotated given number of degrees.
static void	**picture**(double x, double y, **String** filename, double scaledWidth, double scaledHeight) Draws the specified image centered at (*x*, *y*), rescaled to the specified bounding box.
static void	**picture**(double x, double y, **String** filename, double scaledWidth, double scaledHeight, double degrees) Draws the specified image centered at (*x*, *y*), rotated given number of degrees, and rescaled to the specified bounding box.
static void	**point**(double x, double y) Draws a point centered at (*x*, *y*).
static void	**polygon**(double[] x, double[] y) Draws a polygon with the vertices (x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1}).
static void	**rectangle**(double x, double y, double halfWidth, double halfHeight) Draws a rectangle of the specified size, centered at (*x*, *y*).
static void	**save**(**String** filename) Saves the drawing to using the specified filename.
static void	**setCanvasSize**() Sets the canvas (drawing area) to be 512-by-512 pixels.
static void	**setCanvasSize**(int canvasWidth, int canvasHeight) Sets the canvas (drawing area) to be *width*-by-*height* pixels.
static void	**setFont**() Sets the font to the default font (sans serif, 16 point).
static void	**setFont**(**Font** font) Sets the font to the specified value.
static void	**setPenColor**() Sets the pen color to the default color (black).
static void	**setPenColor**(**Color** color) Sets the pen color to the specified color.
static void	**setPenColor**(int red, int green, int blue) Sets the pen color to the specified RGB color.
static void	**setPenRadius**() Sets the pen size to the default size (0.002).
static void	**setPenRadius**(double radius) Sets the radius of the pen to the specified size.
static void	**setScale**() Sets the *x*-scale and *y*-scale to be the default (between 0.0 and 1.0).

static void	**setScale**(double min, double max) Sets both the *x*-scale and *y*-scale to the (same) specified range.
static void	**setXscale**() Sets the *x*-scale to be the default (between 0.0 and 1.0).
static void	**setXscale**(double min, double max) Sets the *x*-scale to the specified range.
static void	**setYscale**() Sets the *y*-scale to be the default (between 0.0 and 1.0).
static void	**setYscale**(double min, double max) Sets the *y*-scale to the specified range.
static void	**show**() Copies offscreen buffer to onscreen buffer.
static void	**show**(int t) Deprecated. replaced by **enableDoubleBuffering()**, **show()**, and **pause(int t)**
static void	**square**(double x, double y, double halfLength) Draws a square of the specified size, centered at (*x*, *y*).
static void	**text**(double x, double y, **String** text) Writes the given text string in the current font, centered at (*x*, *y*).
static void	**text**(double x, double y, **String** text, double degrees) Writes the given text string in the current font, centered at (*x*, *y*) and rotated by the specified number of degrees.
static void	**textLeft**(double x, double y, **String** text) Writes the given text string in the current font, left-aligned at (*x*, *y*).
static void	**textRight**(double x, double y, **String** text) Writes the given text string in the current font, right-aligned at (*x*, *y*).

Methods inherited from class Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

BLACK

```
public static final Color BLACK
```

The color black.

BLUE

```
public static final Color BLUE
```

The color blue.

CYAN

```
public static final Color CYAN
```

The color cyan.

DARK_GRAY

`public static final Color DARK_GRAY`

The color dark gray.

GRAY

`public static final Color GRAY`

The color gray.

GREEN

`public static final Color GREEN`

The color green.

LIGHT_GRAY

`public static final Color LIGHT_GRAY`

The color light gray.

MAGENTA

`public static final Color MAGENTA`

The color magenta.

ORANGE

`public static final Color ORANGE`

The color orange.

PINK

`public static final Color PINK`

The color pink.

RED

`public static final Color RED`

The color red.

WHITE

`public static final Color WHITE`

The color white.

YELLOW

```
public static final Color YELLOW
```

The color yellow.

BOOK_BLUE

```
public static final Color BOOK_BLUE
```

Shade of blue used in *Introduction to Programming in Java*. It is Pantone 300U. The RGB values are approximately (9, 90, 166).

BOOK_LIGHT_BLUE

```
public static final Color BOOK_LIGHT_BLUE
```

Shade of light blue used in *Introduction to Programming in Java*. The RGB values are approximately (103, 198, 243).

BOOK_RED

```
public static final Color BOOK_RED
```

Shade of red used in *Algorithms, 4th edition*. It is Pantone 1805U. The RGB values are approximately (150, 35, 31).

PRINCETON_ORANGE

```
public static final Color PRINCETON_ORANGE
```

Shade of orange used in Princeton University's identity. It is PMS 158. The RGB values are approximately (245, 128, 37).

Method Detail

setCanvasSize

```
public static void setCanvasSize()
```

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.

setCanvasSize

```
public static void setCanvasSize(int canvasWidth,
                                 int canvasHeight)
```

Sets the canvas (drawing area) to be *width*-by-*height* 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.

Parameters:

`canvasWidth` - the width as a number of pixels

`canvasHeight` - the height as a number of pixels

Throws:

`IllegalArgumentException` - unless both `canvasWidth` and `canvasHeight` are positive

setXscale

```
public static void setXscale()
```

Sets the *x*-scale to be the default (between 0.0 and 1.0).

setYscale

```
public static void setYscale()
```

Sets the *y*-scale to be the default (between 0.0 and 1.0).

setScale

```
public static void setScale()
```

Sets the *x*-scale and *y*-scale to be the default (between 0.0 and 1.0).

setXscale

```
public static void setXscale(double min,
                             double max)
```

Sets the *x*-scale to the specified range.

Parameters:

`min` - the minimum value of the *x*-scale

`max` - the maximum value of the *x*-scale

Throws:

`IllegalArgumentException` - if `(max == min)`

`IllegalArgumentException` - if either `min` or `max` is either NaN or infinite

setYscale

```
public static void setYscale(double min,
                             double max)
```

Sets the *y*-scale to the specified range.

Parameters:

`min` - the minimum value of the *y*-scale

`max` - the maximum value of the *y*-scale

Throws:

`IllegalArgumentException` - if `(max == min)`

`IllegalArgumentException` - if either `min` or `max` is either NaN or infinite

setScale

```
public static void setScale(double min,
                            double max)
```

Sets both the *x*-scale and *y*-scale to the (same) specified range.

Parameters:

`min` - the minimum value of the *x*- and *y*-scales

`max` - the maximum value of the *x*- and *y*-scales

Throws:

`IllegalArgumentException` - if `(max == min)`

`IllegalArgumentException` - if either `min` or `max` is either NaN or infinite

clear

```
public static void clear()
```

Clears the screen to the default color (white).

clear

```
public static void clear(Color color)
```

Clears the screen to the specified color.

Parameters:

`color` - the color to make the background

Throws:

`IllegalArgumentException` - if `color` is `null`

getPenRadius

```
public static double getPenRadius()
```

Returns the current pen radius.

Returns:

the current value of the pen radius

setPenRadius

```
public static void setPenRadius()
```

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.

setPenRadius

```
public static void setPenRadius(double 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.

Parameters:

`radius` - the radius of the pen

Throws:

`IllegalArgumentException` - if `radius` is negative, NaN, or infinite

getPenColor

```
public static Color getPenColor()
```

Returns the current pen color.

Returns:

the current pen color

setPenColor

```
public static void setPenColor()
```

Sets the pen color to the default color (black).

setPenColor

```
public static void setPenColor(Color color)
```

Sets the pen color to the specified color.

The predefined pen colors are `StdDraw.BLACK`, `StdDraw.BLUE`, `StdDraw.CYAN`, `StdDraw.DARK_GRAY`, `StdDraw.GRAY`, `StdDraw.GREEN`, `StdDraw.LIGHT_GRAY`, `StdDraw.MAGENTA`, `StdDraw.ORANGE`, `StdDraw.PINK`, `StdDraw.RED`, `StdDraw.WHITE`, and `StdDraw.YELLOW`.

Parameters:

`color` - the color to make the pen

Throws:

`IllegalArgumentException` - if `color` is `null`

setPenColor

```
public static void setPenColor(int red,
                               int green,
                               int blue)
```

Sets the pen color to the specified RGB color.

Parameters:

`red` - the amount of red (between 0 and 255)

`green` - the amount of green (between 0 and 255)

`blue` - the amount of blue (between 0 and 255)

Throws:

`IllegalArgumentException` - if `red`, `green`, or `blue` is outside its prescribed range

getFont

```
public static Font getFont()
```

Returns the current font.

Returns:

the current font

setFont

```
public static void setFont()
```

Sets the font to the default font (sans serif, 16 point).

setFont

```
public static void setFont(Font font)
```

Sets the font to the specified value.

Parameters:

`font` - the font

Throws:

`IllegalArgumentException` - if `font` is `null`

line

```
public static void line(double x0,
                        double y0,
                        double x1,
                        double y1)
```

Draws a line segment between (x_0, y_0) and (x_1, y_1).

Parameters:

`x0` - the *x*-coordinate of one endpoint

`y0` - the *y*-coordinate of one endpoint

`x1` - the *x*-coordinate of the other endpoint

`y1` - the *y*-coordinate of the other endpoint

Throws:

`IllegalArgumentException` - if any coordinate is either NaN or infinite

point

```
public static void point(double x,
                         double y)
```

Draws a point centered at (*x*, *y*). 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.

Parameters:

x - the *x*-coordinate of the point

y - the *y*-coordinate of the point

Throws:

`IllegalArgumentException` - if either `x` or `y` is either NaN or infinite

circle

```
public static void circle(double x,
                          double y,
                          double radius)
```

Draws a circle of the specified radius, centered at (*x*, *y*).

Parameters:

x - the *x*-coordinate of the center of the circle

y - the *y*-coordinate of the center of the circle

`radius` - the radius of the circle

Throws:

`IllegalArgumentException` - if `radius` is negative

`IllegalArgumentException` - if any argument is either NaN or infinite

filledCircle

```
public static void filledCircle(double x,
                                double y,
                                double radius)
```

Draws a filled circle of the specified radius, centered at (*x*, *y*).

Parameters:

x - the *x*-coordinate of the center of the circle

y - the *y*-coordinate of the center of the circle

`radius` - the radius of the circle

Throws:

`IllegalArgumentException` - if `radius` is negative

`IllegalArgumentException` - if any argument is either NaN or infinite

ellipse

```
public static void ellipse(double x,
                           double y,
                           double semiMajorAxis,
                           double semiMinorAxis)
```

Draws an ellipse with the specified semimajor and semiminor axes, centered at (*x*, *y*).

Parameters:

x - the *x*-coordinate of the center of the ellipse

y - the *y*-coordinate of the center of the ellipse

semiMajorAxis - is the semimajor axis of the ellipse

semiMinorAxis - is the semiminor axis of the ellipse

Throws:

IllegalArgumentException - if either semiMajorAxis or semiMinorAxis is negative

IllegalArgumentException - if any argument is either NaN or infinite

filledEllipse

```
public static void filledEllipse(double x,
                                 double y,
                                 double semiMajorAxis,
                                 double semiMinorAxis)
```

Draws a filled ellipse with the specified semimajor and semiminor axes, centered at (*x*, *y*).

Parameters:

x - the *x*-coordinate of the center of the ellipse

y - the *y*-coordinate of the center of the ellipse

semiMajorAxis - is the semimajor axis of the ellipse

semiMinorAxis - is the semiminor axis of the ellipse

Throws:

IllegalArgumentException - if either semiMajorAxis or semiMinorAxis is negative

IllegalArgumentException - if any argument is either NaN or infinite

arc

```
public static void arc(double x,
                       double y,
                       double radius,
                       double angle1,
                       double angle2)
```

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

Parameters:

x - the *x*-coordinate of the center of the circle

y - the *y*-coordinate of the center of the circle

radius - the radius of the circle

angle1 - the starting angle. 0 would mean an arc beginning at 3 o'clock.

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 radius is negative

IllegalArgumentException - if any argument is either NaN or infinite

square

```
public static void square(double x,
                          double y,
                          double halfLength)
```

Draws a square of the specified size, centered at (*x*, *y*).

Parameters:

`x` - the *x*-coordinate of the center of the square

`y` - the *y*-coordinate of the center of the square

`halfLength` - one half the length of any side of the square

Throws:

`IllegalArgumentException` - if `halfLength` is negative

`IllegalArgumentException` - if any argument is either NaN or infinite

filledSquare

```
public static void filledSquare(double x,
                                double y,
                                double halfLength)
```

Draws a filled square of the specified size, centered at (*x*, *y*).

Parameters:

`x` - the *x*-coordinate of the center of the square

`y` - the *y*-coordinate of the center of the square

`halfLength` - one half the length of any side of the square

Throws:

`IllegalArgumentException` - if `halfLength` is negative

`IllegalArgumentException` - if any argument is either NaN or infinite

rectangle

```
public static void rectangle(double x,
                             double y,
                             double halfWidth,
                             double halfHeight)
```

Draws a rectangle of the specified size, centered at (*x*, *y*).

Parameters:

`x` - the *x*-coordinate of the center of the rectangle

`y` - the *y*-coordinate of the center of the rectangle

`halfWidth` - one half the width of the rectangle

`halfHeight` - one half the height of the rectangle

Throws:

`IllegalArgumentException` - if either `halfWidth` or `halfHeight` is negative

`IllegalArgumentException` - if any argument is either NaN or infinite

filledRectangle

```
public static void filledRectangle(double x,
                                   double y,
                                   double halfWidth,
                                   double halfHeight)
```

Draws a filled rectangle of the specified size, centered at (*x*, *y*).

Parameters:

x - the *x*-coordinate of the center of the rectangle

y - the *y*-coordinate of the center of the rectangle

`halfWidth` - one half the width of the rectangle

`halfHeight` - one half the height of the rectangle

Throws:

`IllegalArgumentException` - if either `halfWidth` or `halfHeight` is negative

`IllegalArgumentException` - if any argument is either NaN or infinite

polygon

```
public static void polygon(double[] x,
                           double[] y)
```

Draws a polygon with the vertices (x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1}).

Parameters:

x - an array of all the *x*-coordinates of the polygon

y - an array of all the *y*-coordinates of the polygon

Throws:

`IllegalArgumentException` - unless `x[]` and `y[]` are of the same length

`IllegalArgumentException` - if any coordinate is either NaN or infinite

`IllegalArgumentException` - if either `x[]` or `y[]` is `null`

filledPolygon

```
public static void filledPolygon(double[] x,
                                 double[] y)
```

Draws a filled polygon with the vertices (x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1}).

Parameters:

x - an array of all the *x*-coordinates of the polygon

y - an array of all the *y*-coordinates of the polygon

Throws:

`IllegalArgumentException` - unless `x[]` and `y[]` are of the same length

`IllegalArgumentException` - if any coordinate is either NaN or infinite

`IllegalArgumentException` - if either `x[]` or `y[]` is `null`

picture

```
public static void picture(double x,
                           double y,
                           String filename)
```

Draws the specified image centered at (*x*, *y*). 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.

Parameters:

`x` - the center *x*-coordinate of the image

`y` - the center *y*-coordinate of the image

`filename` - the name of the image/picture, e.g., "ball.gif"

Throws:

`IllegalArgumentException` - if the image filename is invalid

`IllegalArgumentException` - if either `x` or `y` is either NaN or infinite

picture

```
public static void picture(double x,
                           double y,
                           String filename,
                           double degrees)
```

Draws the specified image centered at (*x*, *y*), rotated given number of degrees. The supported image formats are JPEG, PNG, and GIF.

Parameters:

`x` - the center *x*-coordinate of the image

`y` - the center *y*-coordinate of the image

`filename` - the name of the image/picture, e.g., "ball.gif"

`degrees` - is the number of degrees to rotate counterclockwise

Throws:

`IllegalArgumentException` - if the image filename is invalid

`IllegalArgumentException` - if `x`, `y`, `degrees` is NaN or infinite

`IllegalArgumentException` - if `filename` is `null`

picture

```
public static void picture(double x,
                           double y,
                           String filename,
                           double scaledWidth,
                           double scaledHeight)
```

Draws the specified image centered at (*x*, *y*), rescaled to the specified bounding box. The supported image formats are JPEG, PNG, and GIF.

Parameters:

`x` - the center *x*-coordinate of the image

`y` - the center *y*-coordinate of the image

`filename` - the name of the image/picture, e.g., "ball.gif"

`scaledWidth` - the width of the scaled image (in screen coordinates)

`scaledHeight` - the height of the scaled image (in screen coordinates)

Throws:

`IllegalArgumentException` - if either `scaledWidth` or `scaledHeight` is negative

`IllegalArgumentException` - if the image filename is invalid

`IllegalArgumentException` - if `x` or `y` is either NaN or infinite

`IllegalArgumentException` - if `filename` is `null`

picture

```
public static void picture(double x,
                           double y,
                           String filename,
                           double scaledWidth,
                           double scaledHeight,
                           double degrees)
```

Draws the specified image centered at (*x*, *y*), rotated given number of degrees, and rescaled to the specified bounding box. The supported image formats are JPEG, PNG, and GIF.

Parameters:

`x` - the center *x*-coordinate of the image

`y` - the center *y*-coordinate of the image

`filename` - the name of the image/picture, e.g., "ball.gif"

`scaledWidth` - the width of the scaled image (in screen coordinates)

`scaledHeight` - the height of the scaled image (in screen coordinates)

`degrees` - is the number of degrees to rotate counterclockwise

Throws:

`IllegalArgumentException` - if either `scaledWidth` or `scaledHeight` is negative

`IllegalArgumentException` - if the image filename is invalid

text

```
public static void text(double x,
                        double y,
                        String text)
```

Writes the given text string in the current font, centered at (*x*, *y*).

Parameters:

`x` - the center *x*-coordinate of the text

`y` - the center *y*-coordinate of the text

`text` - the text to write

Throws:

`IllegalArgumentException` - if `text` is `null`

`IllegalArgumentException` - if `x` or `y` is either NaN or infinite

text

```
public static void text(double x,
                        double y,
                        String text,
                        double degrees)
```

Writes the given text string in the current font, centered at (*x*, *y*) and rotated by the specified number of degrees.

Parameters:

`x` - the center *x*-coordinate of the text

`y` - the center *y*-coordinate of the text

`text` - the text to write

`degrees` - is the number of degrees to rotate counterclockwise

Throws:

`IllegalArgumentException` - if `text` is `null`

`IllegalArgumentException` - if `x`, `y`, or `degrees` is either NaN or infinite

textLeft

```
public static void textLeft(double x,
                            double y,
                            String text)
```

Writes the given text string in the current font, left-aligned at (*x*, *y*).

Parameters:

`x` - the *x*-coordinate of the text

`y` - the *y*-coordinate of the text

`text` - the text

Throws:

`IllegalArgumentException` - if `text` is `null`

`IllegalArgumentException` - if `x` or `y` is either NaN or infinite

textRight

```
public static void textRight(double x,
                             double y,
                             String text)
```

Writes the given text string in the current font, right-aligned at (*x*, *y*).

Parameters:

`x` - the *x*-coordinate of the text

`y` - the *y*-coordinate of the text

`text` - the text to write

Throws:

`IllegalArgumentException` - if `text` is `null`

`IllegalArgumentException` - if x or y is either NaN or infinite

show

```
@Deprecated
public static void show(int t)
```

Deprecated. replaced by `enableDoubleBuffering()`, `show()`, and `pause(int t)`
Copies the offscreen buffer to the onscreen buffer, pauses for t milliseconds and enables double buffering.

Parameters:

`t` - number of milliseconds

pause

```
public static void pause(int t)
```

Pauses for t milliseconds. This method is intended to support computer animations.

Parameters:

`t` - number of milliseconds

show

```
public static void show()
```

Copies offscreen buffer to onscreen buffer. There is no reason to call this method unless double buffering is enabled.

enableDoubleBuffering

```
public static void enableDoubleBuffering()
```

Enables double buffering. All subsequent calls to drawing methods such as `line()`, `circle()`, and `square()` will be deferred until the next call to show(). Useful for animations.

disableDoubleBuffering

```
public static void disableDoubleBuffering()
```

Disables double buffering. All subsequent calls to drawing methods such as `line()`, `circle()`, and `square()` will be displayed on screen when called. This is the default.

save

```
public static void save(String filename)
```

Saves the drawing to using the specified filename. The supported image formats are JPEG and PNG; the filename suffix must be `.jpg` or `.png`.

Parameters:

`filename` - the name of the file with one of the required suffixes

Throws:

`IllegalArgumentException` - if `filename` is `null`

actionPerformed

```
public void actionPerformed(ActionEvent e)
```

This method cannot be called directly.

Specified by:

`actionPerformed` in interface `ActionListener`

isMousePressed

```
public static boolean isMousePressed()
```

Returns true if the mouse is being pressed.

Returns:

`true` if the mouse is being pressed; `false` otherwise

mousePressed

```
@Deprecated
public static boolean mousePressed()
```

Deprecated. replaced by `isMousePressed()`
Returns true if the mouse is being pressed.

Returns:

`true` if the mouse is being pressed; `false` otherwise

mouseX

```
public static double mouseX()
```

Returns the *x*-coordinate of the mouse.

Returns:

the *x*-coordinate of the mouse

mouseY

```
public static double mouseY()
```

Returns the *y*-coordinate of the mouse.

Returns:

y-coordinate of the mouse

mouseClicked

`public void mouseClicked(MouseEvent e)`

This method cannot be called directly.

Specified by:

`mouseClicked` in interface `MouseListener`

mouseEntered

`public void mouseEntered(MouseEvent e)`

This method cannot be called directly.

Specified by:

`mouseEntered` in interface `MouseListener`

mouseExited

`public void mouseExited(MouseEvent e)`

This method cannot be called directly.

Specified by:

`mouseExited` in interface `MouseListener`

mousePressed

`public void mousePressed(MouseEvent e)`

This method cannot be called directly.

Specified by:

`mousePressed` in interface `MouseListener`

mouseReleased

`public void mouseReleased(MouseEvent e)`

This method cannot be called directly.

Specified by:

`mouseReleased` in interface `MouseListener`

mouseDragged

`public void mouseDragged(MouseEvent e)`

This method cannot be called directly.

Specified by:

`mouseDragged` in interface `MouseMotionListener`

mouseMoved

```
public void mouseMoved(MouseEvent e)
```

This method cannot be called directly.

Specified by:

`mouseMoved` in interface `MouseMotionListener`

hasNextKeyTyped

```
public static boolean hasNextKeyTyped()
```

Returns true if the user has typed a key (that has not yet been processed).

Returns:

`true` if the user has typed a key (that has not yet been processed by `nextKeyTyped()`; `false` otherwise

nextKeyTyped

```
public static char nextKeyTyped()
```

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 `hasNextKeyTyped()` to ensure that there is a next key to process. This method returns a Unicode character corresponding to the key typed (such as `'a'` or `'A'`). It cannot identify action keys (such as F1 and arrow keys) or modifier keys (such as control).

Returns:

the next key typed by the user (that your program has not already processed).

Throws:

`NoSuchElementException` - if there is no remaining key

isKeyPressed

```
public static boolean isKeyPressed(int keycode)
```

Returns true if the given key is being pressed.

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 `KeyEvent` for a description of key codes.

Parameters:

`keycode` - the key to check if it is being pressed

Returns:

`true` if `keycode` is currently being pressed; `false` otherwise

keyTyped

`public void keyTyped(KeyEvent e)`

This method cannot be called directly.

Specified by:

`keyTyped` in interface `KeyListener`

keyPressed

`public void keyPressed(KeyEvent e)`

This method cannot be called directly.

Specified by:

`keyPressed` in interface `KeyListener`

keyReleased

`public void keyReleased(KeyEvent e)`

This method cannot be called directly.

Specified by:

`keyReleased` in interface `KeyListener`

main

`public static void main(String[] args)`

Test client.

Parameters:

`args` - the command-line arguments