Chapter 12

Simple Graphics Programming

Graphical applications are more appealing to users than text-based programs. When designed properly, graphical applications are more intuitive to use. From the developers perspective, graphical programs provide an ideal opportunity to apply object-oriented programming concepts. Java provides a wealth of classes for building graphical user interfaces (GUIs). Because of the power and flexibility provided by these classes experienced developers can use them to create sophisticated graphical applications. On the other hand, also because of the power and flexibility of these graphical classes, writing even a simple graphics program can be daunting for those new to programming—especially those new to graphics programming. For those experienced in building GUIs, Java simplifies the process of developing good GUIs. Good GUIs are necessarily complex. Consider just two issues:

- Graphical objects must be positioned in an attractive manner even when windows are resized or the application must run on both a PC (larger screen) and a PDA or cell phone (much smaller screen). Java graphical classes simplify the programmer's job of laying out of screen elements within an application.
- The user typically can provide input in several different ways: via mouse clicks, menu, dialog boxes, and keystrokes. The program often must be able to respond to input at any time, not at some predetermined place in the program's execution. For example, when a statement calling `nextInt()` on a `Scanner` object (§ 8.3) is executed, the program halts and waits for the user's input; however, in a graphical program the user may select any menu item at any time. We say that graphics programs are *event driven* instead of program driven. Java's event classes readily support event-driven program development.

While Java's infrastructure is a boon to seasoned GUI programmers, at this time Java has no standard graphics classes that make graphics programming more accessible to beginners. The unfortunate result is that beginners must learn a number of different principles (including Java's layout management and event model), before writing even the simplest interactive graphical programs.

In this chapter we introduce some simplified graphics classes that beginners can use to write interactive graphical programs. These classes are built from standard Java classes, but they insulate the programmer from much of the complexity of Java GUI development.

12.1 2D Graphics Concepts

A *graphic* is an image drawn on a display device, usually the screen. The smallest piece of a graphic is called a *pixel*, short for *picture element*. The size of a pixel is fixed by the hardware resolution of the display device. A graphic is

rendered on a rectangular drawing surface. Modern operating systems use a window-based interface, so the drawing surface for a graphical application is usually the area inside the frame of a window assigned to that application. Each rectangular drawing surface has a coordinate system. The coordinate system is slightly different from the Cartesian coordinate system used in mathematics:

- the origin, $(0,0)$, is located at the left-top corner of the rectangular region, and
- the *y* axis points down, meaning that the *y* values increase as you go down.

Figure 12.1 illustrates:

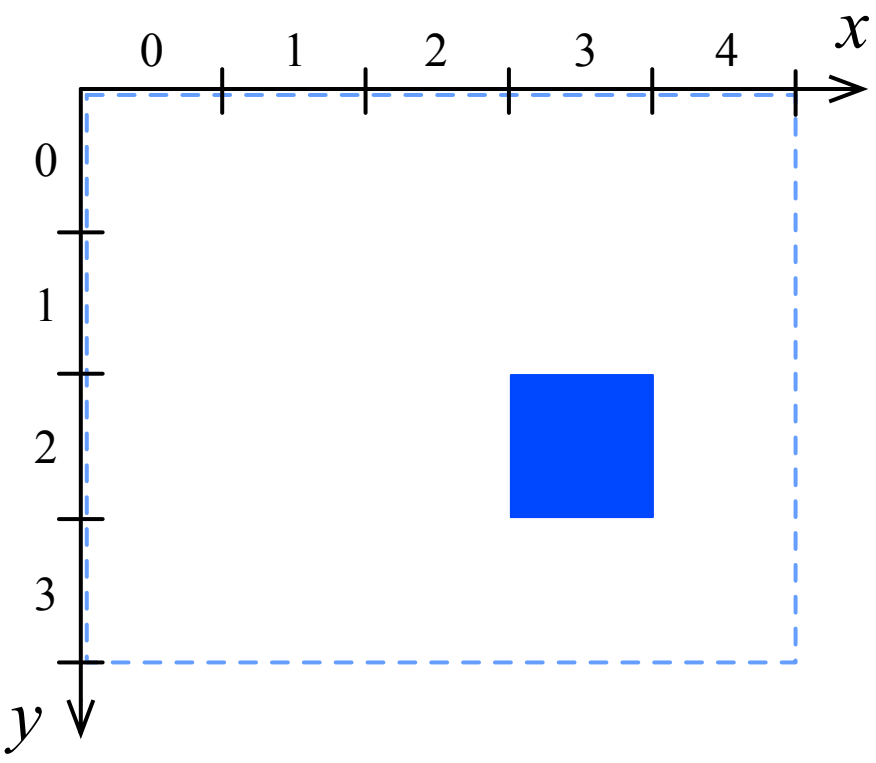

Figure 12.1: The graphics coordinate system. The small square is a highly magnified view of a pixel at coordinate $(3,2)$. The dotted frame represents the bounds of the rectangular drawing surface which is five pixels wide and four pixels tall.

12.2 The `Viewport` Class

In our simplified graphics classes a graphical window is represented by an object of type `Viewport`.[1] Graphics can be displayed within viewports, and viewports can receive input events from the user, such as mouse movement and clicking. The following interactive sequence creates and manipulates a simple viewport:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> import edu.southern.computing.oopj.Viewport;
```

[1] The name `Viewport` was chosen to be especially distinct from the standard Java window types: `Frame` and `JFrame`. `Viewport` is implemented as a subclass of `javax.swing.JFrame`.


```
> w = new Viewport("Very Simple Window", 100, 100, 300, 200);
> w.setSize(400, 550);
> w.setLocation(50, 100);
```

In this brief interactive sequence:

- The first statement imports the necessary class definition. Since the package name does not begin with `java...`, we know immediately that this is not a standard class.

- Variable `w` is assigned to a new `Viewport` object. The parameters to the constructor are, in order:

  – the string title to appear in the window's (viewport's) title bar,

  – the *x* coordinate of the left-top corner of the viewport,

  – the *y* coordinate of the left-top corner of the viewport,

  – the width of the viewport in pixels, and

  – the height of the viewport in pixels.

- The `setSize()` method resizes the window to the specified width and height. The window's left top corner does not move.

- The `setLocation()` method repositions the window so that its left top corner is moved to the specified (x,y) location. The window's size is unchanged.

Observe that in addition to calling the `setSize()` and `setLocation()` methods, the user can reposition and resize the window using the mouse or other pointing device.

The `Viewport` class has a `draw()` method that determines the visual contents of a viewport. The default `draw()` method does nothing, hence our empty viewport in the example above. In order to draw within a viewport, we must create a subclass of `Viewport` and override the `draw()` method.

A viewport provides methods that can be used within the `draw()` method to render primitive shapes. The shapes include rectangles, ellipses, polygons, lines, points, and strings. Table 12.1 lists some of the more frequently used methods:

Some Methods of the drawingprimitives Class
`void setColor(Color c)` Sets the current drawing color to `c`. `c` can be one of `BLACK`, `WHITE`, `RED`, `BLUE`, color constants defined for rectangular drawing surfaces.
`void drawLine(int x1, int y1, int x2, int y2, Color c)` Draws a one-pixel wide line from (`x1`,`y1`) to (`x2`,`y2`). If the last parameter is omitted, the current drawing color is used.
`void drawPoint(int x, int y, Color c)` Plots a point (a single pixel) at a given (x,y) position. If the last parameter is omitted, the current drawing color is used.
`void drawRectangle(int left, int top, int width, int height)` Draws the outline of a rectangle given the position of its left-top corner and size.
`void fillRectangle(int left, int top, int width, int height, Color c)` Renders a solid of a rectangle given the position of its left-top corner and size. If the last parameter is omitted, the current drawing color is used.
`void drawOval(int left, int top, int width, int height)` Draws the outline of an ellipse given the position of the left-top corner and size of its bounding rectangle.
`void fillOval(int left, int top, int width, int height, Color c)` Renders a solid of a ellipse given the position of the left-top corner and size of its bounding rectangle. If the last parameter is omitted, the current drawing color is used.
`void fillPolygon(/*List of coordinates*/)` Renders a solid polygon with the current drawing color. The parameters are (x,y) pairs of integers specifying the polygon's vertices (corners).

Table 12.1: A subset of the drawing methods provided by the `Viewport` class, a rectangular drawing surface. These methods **should not be used outside** of the `draw()` method.

More complex pictures can be drawn by combining several primitive shapes. These drawing methods **should not be used outside** of the `draw()` method. Furthermore, code you write **should not attempt to call** the `draw()` method directly. It is the responsibility of the window manager to call your viewport's `draw()` method.[2]

The `Viewport` class defines a number of public constant color objects that affect drawing. These colors include `RED`, `BLUE`, `GREEN`, `YELLOW`, `CYAN`, `MAGENTA`, `GRAY`, `ORANGE`, `PINK`, `LIGHT_GRAY`, `DARK_GRAY`, `WHITE`, `BLACK`, and `TRANSPARENT`. These are `java.awt.Color` objects. If the predefined colors do not meet your need, you can make your own color object by specifying the proper combination of red-green-blue (RGB) primary color values. The primary color values are integers than can range from 0 to 255. Black is $(0,0,0)$ representing no contribution by any of the primary colors; white is $(255,255,255)$ representing all primary colors contributing fully. The following statement

```
Color lightGreen = new Color(100, 255, 100);
```

creates a color object with red contributing $\frac{100}{255}$, green contributing fully ($\frac{255}{255}$), and contributing $\frac{100}{255}$. This results in a lighter shade of green than `Viewport.GREEN` which has an RGB combination of $(0,255,0)$.

The background color of a viewport can be set with `setBackground()` that accepts a single `Color` parameter.

To see how this all works, let us draw a static picture of a traffic light. Our simple graphical light will be composed of a rectangular gray frame and three circular lamps of the appropriate colors. Figure 12.2 shows our

[2] Actually the window manager calls the viewport's `repaint()` method which through a chain of standard graphical painting methods eventually calls the viewport's `draw()` method.

graphical design. The numbers shown are pixel values. The rectangular frame can be rendered with the viewport's

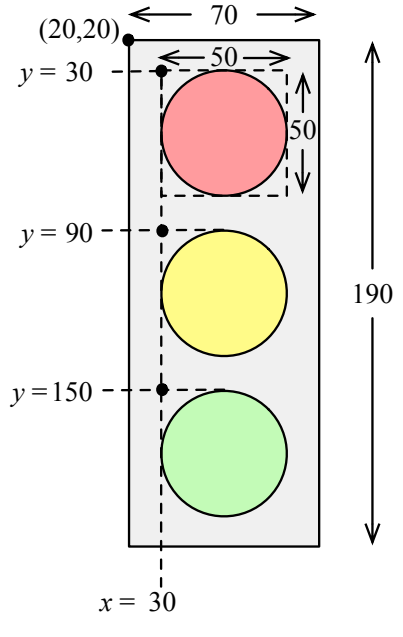

Figure 12.2: Graphical traffic light design.

`fillRectangle()` method. The `fillRectancle()` method expects an (x,y) value for the rectangle's left-top corner, a width value, a height value, and a color. According to the figure we would use the call

```
fillRectangle(20, 20, 70, 190, GRAY);
```

Each lamp is an ellipse with a square bounding box (width = height, effectively drawing a circle). The top (red) lamp's bounding box is offset 10 pixels horizontally and vertically from the frame's left-top corner, and its diameter is 50 pixels. The following statement is exactly what we need:

```
fillOval(30, 30, 50, 50, RED);
```

The remaining lamps are the same size (50×50) and offset the same vertically (30 pixels). They obviously need bigger vertical offsets. If we want 10 pixels between each lamp, and the lamps are 50 pixels across, we need to add 60 to the *y* value of the previous lamp's drawing statement. The statements

```
fillOval(30, 90, 50, 50, YELLOW);
fillOval(30, 150, 50, 50, GREEN);
```

will work nicely.

The four drawing statements we derived above must be placed in the `draw()` method of a `Viewport` subclass. `DrawTrafficLight` (12.1) works nicely:

```
import edu.southern.computing.oopj.Viewport;

public class DrawTrafficLight extends Viewport {
    public DrawTrafficLight() {
        super("Traffic Light", 100, 100, 50, 270);
```



```
    }
    public void draw() {
        fillRectangle(20, 20, 70, 190, GRAY);
        fillOval(30, 30, 50, 50, RED);
        fillOval(30, 90, 50, 50, YELLOW);
        fillOval(30, 150, 50, 50, GREEN);
    }
}
```

Listing 12.1: DrawTrafficLight—Renders a graphical traffic light

The following interactive session draws the light:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> new DrawTrafficLight();
```

DrawStar (12.2) uses the drawPolygon() method to draw a five-pointed star:

```
import edu.southern.computing.oopj.Viewport;

public class DrawStar extends Viewport {

    public DrawStar() {
        super("Star", 100, 100, 200, 200);
        setBackground(WHITE);
    }

    public void drawStar(int xCenter, int yCenter) {
        //  Code adapted from http://www.research.att.com/
        //                    sw/tools/yoix/doc/graphics/
        //                    pointInPolygon.html
        drawPolygon(xCenter +  0,  yCenter - 50,
                    xCenter + 29,  yCenter + 40,
                    xCenter - 47,  yCenter - 15,
                    xCenter + 47,  yCenter - 15,
                    xCenter - 29,  yCenter + 40);
    }

    public void draw() {
        drawStar(getWidth()/2, getHeight()/2);

    }

    public static void main(String[] args) {
        new DrawStar();
    }
}
```


Listing 12.2: `DrawStar`—draws a five-pointed star

As in the case of `DrawTrafficLight` (12.1), all drawing is performed within the overridden `draw()` method.

12.3 Event Model

Users interact differently with graphical programs than they do with text-based programs. A text-based program predetermines when input is needed. For example, suppose `scan` is a `Scanner` object. To receive an integer value from the user the statement

```
int value = scan.nextInt();
```

causes the program's execution to come to a complete halt until the user provides the requested value.

In a graphical program, the user may move the mouse pointer over particular portion of the window and then click and drag the mouse. Instead, the user may select an item from the menu bar, or right click the mouse button to bring up a context-sensitive popup menu. Perhaps user presses a keyboard shortcut key like Ctrl S. Each of these activities, such as moving the mouse, pressing the mouse button, typing a key, or selecting a menu item, is called an *event*. The window manager monitors the program's execution watching for events to transpire. When an event occurs, the window manager notifies the running program. The program in turn either responds to or ignores the event.

Our viewport objects can respond to mouse and key events. The window manager notifies our viewport object that an event has occurred by calling a method that corresponds to that event. Viewport objects currently can handle the events shown in Table 12.2. By default, these methods do nothing. In order to allow your viewport to respond

Event	Method
Mouse button pressed	`mousePressed()`
Mouse button released	`mouseReleased()`
Mouse button clicked	`mouseClicked()`
Mouse pointer moved over viewport from outside	`mouseEntered()`
Mouse pointer moved out of viewport	`mouseExited()`
Mouse moved while button depressed	`mouseDragged()`
Mouse moved with no buttons depressed	`mouseMoved()`
Key typed	`keyTyped()`

Table 12.2: Viewport events and their corresponding methods

to events in ways appropriate for your application, subclass `Viewport` and override the corresponding methods. `SimpleResponder` (12.3) illustrates how this is done:

```
import edu.southern.computing.oopj.Viewport;

public class SimpleResponder extends Viewport {
    public SimpleResponder() {
        super("Very Simple Responder", 100, 100, 400, 300);
```



```
    }
    //  The window manager calls this method when the user depresses
    //  the mouse button when the mouse pointer is over the viewport
    public void mousePressed() {
        System.out.println("Mouse is at (" + getMouseX() + ","
                           + getMouseY() + ")");
    }
    //  The window manager calls this method when the user types a
    //  a key when the viewport has the focus
    public void keyTyped() {
        System.out.println("'" + getKeyTyped() + "' typed");
    }
}
```

Listing 12.3: `SimpleResponder`—monitors two kinds of events—mouse button presses and keystrokes

In the Interactions pane simply create an instance of `SimpleResponder`. See what happens when you click the mouse and press keys in the viewport.

If your custom viewport class does not override an event method, it in effect ignores that event. In reality the window manager calls the appropriate method anyway, but the empty body does nothing. It is common for applications not to respond to various events, so often your custom viewport will override few, if any, of the event methods.

`SimpleResponder` (12.3) reveals several other viewport methods that are valuable when writing event handlers:

- `getMouseX()`—returns the *x* coordinate of the mouse pointer's location when the mouse event occurred,
- `getMouseY()`—returns the *y* coordinate of the mouse pointer's location when the mouse event occurred, and
- `getKeyTyped()`—returns the character corresponding to the key typed.

These methods need not be used at all; sometimes it is sufficient to know that the mouse button was pressed, and your application does not care where the mouse was at the time. Similarly, sometimes your application needs to know when any key is typed and does not care about which particular key was typed. `InteractiveTrafficLightViewport` (12.4) shows how the `mouseClicked()` method can be overridden using neither `getMouseX()` nor `getMouseY()`. It makes use of `TrafficLightModel` (10.1).

```
import edu.southern.computing.oopj.Viewport;

public class InteractiveTrafficLightViewport extends Viewport {
    //  The traffic light model controls the state of the light
    private TrafficLightModel model;

    public InteractiveTrafficLightViewport() {
        super("Traffic Light---Click to change", 100, 100, 50, 270);
        model = new TrafficLightModel(TrafficLightModel.STOP);
    }
```



```
    //  Conditionally illuminate lamps based on the state of the
    //  traffic light model
    public void draw() {
        fillRectangle(20, 20, 70, 190, GRAY);
        int state = model.getState();
        if (state == TrafficLightModel.STOP) {
            fillOval(30, 30, 50, 50, RED);
        } else {
            fillOval(30, 30, 50, 50, BLACK);
        }
        if (state == TrafficLightModel.CAUTION) {
            fillOval(30, 90, 50, 50, YELLOW);
        } else {
            fillOval(30, 90, 50, 50, BLACK);
        }
        if (state == TrafficLightModel.GO) {
            fillOval(30, 150, 50, 50, GREEN);
        } else {
            fillOval(30, 150, 50, 50, BLACK);
        }
    }

    //  The window manager calls this method when the user clicks
    //  the mouse button when the mouse pointer is over the viewport
    public void mouseClicked() {
        model.change();
    }
}
```

Listing 12.4: `InteractiveTrafficLightViewport`—a simple interactive graphical traffic light

In `InteractiveTrafficLightViewport` (12.4), the user simply clicks the mouse over the window to change the traffic light.

12.4 Anonymous Inner Classes

In Java, a class can be defined within another class's definition. The enclosed class is called an *inner class*, and the enclosing class is called the *outer class*. While it is legal and sometimes beneficial to do so, we often will not need to define such named nested classes. One related aspect of nested classes is extremely useful, however. Sometimes it is convenient to create an instance of a subclass without going to the trouble of defining the separate, named subclass. For example, recall `SimpleResponder` (12.3). Clearly `SimpleResponder` is a subclass of `Viewport`, and this subclassing is necessary so we can override various methods to achieve the interactive viewport we want. We can write a main program that uses `SimpleResponder` (12.3) as shown in `UsingSimpleResponder` (12.5):

```
import edu.southern.computing.oopj.Viewport;
```



```
public class UsingSimpleResponder {
    public static void main(String[] args) {
        new SimpleResponder();
    }
}
```

Listing 12.5: `UsingSimpleResponder`—uses the `SimpleResponder` class.

`SimpleResponder` is a simple straightforward extension of the `Viewport` class. If we never need to use the `SimpleResponder` class ever again, it seems wasteful to go to the trouble to define a separate class which requires the creation of an associated `SimpleResponder.java` source file. Observe that a `SimpleResponder` object is a straightforward extension of a plain `Viewport` object, as only two methods are overridden. Ideally, we should be able to create a `Viewport` object that has a little added functionality over the stock viewport without going to the trouble of defining a new named class. Java's *anonymous inner class* feature allows us to do so.

Creating an anonymous inner class is easy. It looks like a combination of object creation and class definition, and indeed that is exactly what it is. `AnonymousSimpleResponder` (12.6) avoids defining a separate named subclass of `Viewport`:

```
import edu.southern.computing.oopj.Viewport;

public class AnonymousSimpleResponder {
    public static void main(String[] args) {
        new Viewport("Anonymous Simple Responder",
                     100, 100, 400, 300) {
            //  What to do when the mouse is pressed
            public void mousePressed() {
                System.out.println("Mouse is at (" + getMouseX() + ","
                                   + getMouseY() + ")");
            }
            //  The window manager calls this method when the user types a
            //  a key when the viewport has the focus
            public void keyTyped() {
                System.out.println("'" + getKeyTyped() + "' typed");
            }
        };
    }
}
```

Listing 12.6: `AnonymousSimpleResponder`—avoids the use of the `SimpleResponder` class.

It initially appears that we are creating a simple `Viewport` object with the `new` operator, but the statement does not end there; in fact, this statement does not end until the semicolon after the close curly brace on the third-to-the-last line. What comes in between looks like the body of a class definition, and indeed it is. It is the definition of

the anonymous inner class. One source file is created (`AnonymousSimpleResponder.java`), but the compiler creates two bytecode files: `AnonymousSimpleResponder.class` and `AnonymousSimpleResponder$1.class`. The second file contains the anonymous inner class code.

The nice thing about inner classes is they have access to the methods and variables of their enclosing outer classes. The next section shows how this capability is very useful.

12.5 A Popup Menu Class

Another convenience class, `ContextMenu` enables programmers to add a popup menu to a viewport. `ContextMenu` objects are very simple—the class has one constructor and one method. The constructor accepts any number of strings representing menu items. The `handler()` method accepts a single string parameter. A viewport's popup menu is set via `setContextMenu()`. When the user clicks the right mouse button (in Mac OS X, Ctrl click triggers the popup menu) the popup menu appears. If the user selects an item from the menu, the event manager calls the `handler()` method of the menu, passing the name of the selected menu item. A conditional statement within `handler()` decides what action should be taken.

`SimpleMenuExample` (12.7) shows how the process works:

```
import edu.southern.computing.oopj.Viewport;
import edu.southern.computing.oopj.ContextMenu;

public class SimpleMenuExample {
    private static int value = 0;
    public static void main(String[] args) {
        Viewport window = new Viewport("Useless Default Menu",
                                       100, 100, 400, 300) {
            public void draw() {
                drawString(Integer.toString(value), 50, 80);
                if (value > 0) {
                    fillRectangle(50, 120, 5*value, 20,
                                  Viewport.BLUE);
                } else {
                    fillRectangle(50 + 5*value, 120, -5*value,
                                  20, Viewport.RED);
                }
            }
        };

        window.setContextMenu(new ContextMenu("Increase",
                                              "Decrease",
                                              "Quit") {
            public void handler(String item) {
                if (item.equals("Increase")) {
                    value++;
                } else if (item.equals("Decrease")) {
                    value--;
                } else if (item.equals("Quit")) {
                    System.exit(0);
                }
```



```
                }
            });
        }
}
```

Listing 12.7: `SimpleMenuExample`—uses a popup menu to modify the displayed number.

`SimpleMenuExample` (12.7) displays two pieces of information:

- a number representing the current value of the `value` variable and
- a bar with a length that reflects `value`'s value. (What happens when you decrease value to below zero?)

Study the structure of `SimpleMenuExample` (12.7) very carefully. It uses two anonymous inner classes:

- The `window` variable is assigned an instance of a class derived from `Viewport`.
- The parameter of the invocation of `setContextMenu()` is an instance of a subclass of `ContextMenu`.

An anonymous inner class can define instance variables just like any other class. Anonymous inner classes cannot have constructors, since by definition a constructor has the same name as the class, but the class has no name!

12.6 Summary

The `edu.southern.computing.oopj.Viewport` class is a subclass of the standard class `javax.swing.JPanel`. The Sun Java documentation and many books cover the `JPanel` class well. The following lists some useful methods in the `Viewport` class and also includes `JPanel` methods that are commonly used in simple graphics programs:

Constructor

- `Viewport(String title, int x, int y, int width, int height)`

  Creates a new `width`×`height` viewport object with `title` in its titlebar, and left-top corner at (x,y).

Manipulation

- `int getX()`

  Returns the *x* coordinate of the viewport's left-top corner.

- `int getY()`

  Returns the *y* coordinate of the viewport's left-top corner.

- `int getWidth()`

  Returns the width of the viewport in pixels.

- `int getHeight()`

  Returns the height of the viewport in pixels.

- `int setLocation(int x, int y)`

  Sets the left-top corner of the viewport to location (x,y). The viewport's size is unchanged.

- `int setSize(int w, int h)`

  Sets the width and height of the viewport to $w \times h$. The viewport's left-top corner location is unchanged.

Mouse Events

- `void mousePressed()`

  Called by the window manager when the user presses the left mouse button when the mouse pointer is within the viewport.

- `void mouseReleased()`

  Called by the window manager when the user releases the left mouse button when the mouse pointer is within the viewport.

- `void mouseClicked()`

  Called by the window manager when the user presses and releases the left mouse button when the mouse pointer is within the viewport.

- `void mouseEntered()`

  Called by the window manager when the user moves the mouse cursor into the viewport from outside the viewport.

- `void mouseExited()`

  Called by the window manager when the user moves the mouse cursor out of the viewport from inside the viewport.

- `void mouseMoved()`

  Called by the window manager when the user moves the mouse cursor within the viewport while no mouse buttons are depressed.

- `void mouseDragged()`

  Called by the window manager when the user moves the mouse cursor within the viewport while a mouse button is depressed.

- `int getMouseX()`

  Returns the *x* coordinate of the mouse cursor during the previous mouse event.

- `int getMouseY()`

  Returns the *y* coordinate of the mouse cursor during the previous mouse event.

Keyboard Events

- `void keyTyped()`

  Called by the window manager when the user types a key when the window has keyboard focus.

- `char getKeyTyped()`

  Returns the character typed during the previous keyboard event.

Graphics

- `void draw()`

  Called by the window manager when the contents of the viewport needs to be displayed. All of the following methods are ordinarily called by programmer-written code within the `draw()` method.

- `void setColor(Color color)`

  Sets the current drawing color to `color`. Several predefined constants are available: `BLACK`, `WHITE`, `RED`, `BLUE`, `GREEN`, `YELLOW`, `CYAN`, `GRAY`, `DARK_GRAY`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, and `TRANSPARENT`.

- `void drawPoint(int x, int y, Color color)`

  An overloaded method; parameter `color` is optional. Draws a single pixel point at (x,y) with color `color`. If the `color` parameter is omitted, the current drawing color in used.

- `void drawLine(int x1, int y1, int x2, int y2, Color color)`

  An overloaded method; parameter `color` is optional. Draws a line connecting points `(x1,y1)` and `(x2,y2)`. with color `color`. If the `color` parameter is omitted, the current drawing color in used.

- `void drawRectangle(int x, int y, int width, int height, Color color)`

  An overloaded method; parameter `color` is optional. Draws a `width`×`height` rectangle with left-top corner at (x,y). The rectangle's color is `color`. If the `color` parameter is omitted, the current drawing color in used.

- `void fillRectangle(int x, int y, int width, int height, Color color)`

  An overloaded method; parameter `color` is optional. Draws a `width`×`height` rectangle with left-top corner at (x,y). The interior of the rectangle is filled completely with the color specified by `color`. If the `color` parameter is omitted, the current drawing color in used.

- `void drawOval(int x, int y, int width, int height, Color color)`

  An overloaded method; parameter `color` is optional. Draws an ellipse totally contained by a `width`×`height` bounding rectangle with left-top corner at (x,y). The ellipse's color is `color`. If the `color` parameter is omitted, the current drawing color in used.

- `void fillOval(int x, int y, int width, int height, Color color)`

  An overloaded method; parameter `color` is optional. Draws an ellipse totally contained by a `width`×`height` bounding rectangle with left-top corner at (x,y). The ellipse's interior is filled completely with the color specifed by `color`. If the `color` parameter is omitted, the current drawing color in used.

- `void drawPolygon(`*list of integers*`)`

  Draws a polygon with vertices at the locations specified in the parameter list. The first integer is the x coordinate of the first vertex, the second parameter is the y coordinate of the first vertex, the third parameter is the x coordinate of the second vertex, the fourth parameter is the y coordinate of the second vertex, etc. The current drawing color in used.

- `void fillPolygon(`*list of integers*`)`

  Works like `drawPolygon()`, but a filled polygon is drawn instead of an outline. The current drawing color in used.

- `void drawString(String message, int x, int y)`

  Draws a string at location (x,y). The current drawing color is used.

12.7 Exercises

1. Make a new interactive graphical traffic light that uses `TurnLightModel` ( 11.1) as its model. Your new class should properly display a green turn arrow.

