1 Handling Input from the Mouse

Next, we will learn how to handle mouse events. Mouse events come in two groups. The first group is mouse events, which covers the mouse entering a component, exiting a component, a mouse button being pressed, a mouse button being released, and a mouse button being clicked. The second group is mouse motion events, which covers the mouse being moved or dragged. The following two sections will cover these groups.

2 Mouse Events

The relevant interface in this case (which must be implemented by any class which is interested in handling mouse events) is `MouseListener`. The event generated (and gets passed to the interface) is `MouseEvent`. The method for attaching a listener to a `Component` is, not surprisingly, `addMouseListener()`. Now to the details:

2.1 The `MouseListener` Interface

This interface contains five methods, explained below:

```
public void mouseClicked(MouseEvent e)

       Invoked when the mouse has been clicked on a component.

public void mousePressed(MouseEvent e)

       Invoked when a mouse button has been pressed on a component.

public void mouseReleased(MouseEvent e)

       Invoked when a mouse button has been released on a component.

public void mouseEntered(MouseEvent e)

       Invoked when the mouse enters a component.

public void mouseExited(MouseEvent e)

       Invoked when the mouse exits a component.
```

2.2 the `MouseEvent` Class

The `MouseEvent` class carries information about the mouse event that occured. The methods provided are as follows:

```
public int getX()

       Returns the horizontal x position of the event relative to the
       source component.
```

public int getY()

Returns the vertical y position of the event relative to the source component.

getPoint

public Point getPoint()

Returns the x,y position of the event relative to the source component.

Returns:
a Point object containing the x and y coordinates relative to the source component

public void translatePoint(int x, int y)

Translates the event's coordinates to a new position by adding specified x (horizontal) and y (veritcal) offsets.

Parameters:
x - the horizontal x value to add to the current x coordinate position
y - the vertical y value to add to the current y coordinate position

public int getClickCount()

Return the number of mouse clicks associated with this event.
Returns:
integer value for the number of clicks

public boolean isPopupTrigger()

Returns whether or not this mouse event is the popup-menu trigger event for the platform.

public String paramString()

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Overrides:
paramString in class ComponentEvent
Returns:
a string identifying the event and its attributes

In addition to these methods, it is also possible to figure out *which mouse button* a given event is associated with. This is handled through the `getModifiers()` method inherited from `InputEvent`.

```
public int getModifiers()

       Returns the modifiers flag for this event.

public static final int BUTTON1_MASK

       The mouse button1 modifier constant.

public static final int BUTTON2_MASK

       The mouse button2 modifier constant.

public static final int BUTTON3_MASK

       The mouse button3 modifier constant.
```

3 Mouse Motion Events

In order to handle mouse motion events, one has to implement the `MouseMotionListener` interface. When a mouse motion event occurs, an object of class `MouseEvent` is generated, and passed to the interface. Once again, one registers to listen to mouse motion events generated by a `Component` using the `addMouseMotionListener()` method.

Since the event object is the same as the one used for mouse events, we will examine the interface only:

3.1 The `MouseMotionListener` Interface

This interface has two methods only, which are as follows:

```
public void mouseDragged(MouseEvent e)

       Invoked when a mouse button is pressed on a component and then
       dragged. Mouse drag events will continue to be delivered to the
       component where the first originated until the mouse button is
       released (regardless of whether the mouse position is within
       the bounds of the component).

public void mouseMoved(MouseEvent e)

       Invoked when the mouse button has been moved on a component
       (with no buttons down).
```

4 Handling Input from the Keyboard

Most graphical user interface (GUI) components which are supposed to do keyboard handling (`TextFields`, `TextAreas`) do their jobs automatically – you do not need to low-level keyboard events in order to make them work. Just like you did not need the low-level mouse events to make a `Button` work.

In this case, the relevant interface is `KeyListener`. The event that gets passed to the interface when a keyboard event occurs is `KeyEvent`. The method to register in order to receive `KeyEvents` is `addKeyListener`. The details are as follows:

4.1 The `KeyListener` Interface

This interface has three methods, as defined below:

```
public void keyTyped(KeyEvent e)

       Invoked when a key has been typed. This event occurs when a key
       press is followed by a key release.

public void keyPressed(KeyEvent e)

       Invoked when a key has been pressed.

public void keyReleased(KeyEvent e)

       Invoked when a key has been released.
```

The `keyPressed()` and `keyReleased()` methods are called when a key is pressed or released, respectively. These methods are platform-dependent, and are fired whether the key being pressed or released produces a printable character or not. For instance, they will be fired for function keys, the **HELP** key or any other key on the keyboard. The `keyTyped()` method is invoked only when a specific unicode character is typed from the keyboard.

4.2 The `KeyEvent` Class

The `KeyEvent` class carries information about the key event that just occurred. The Java documentation gives the following important information:

```
"Key typed" events are higher-level and generally do not depend on the
platform or keyboard layout. They are generated when a character is
entered, and are the preferred way to find out about character
input. In the simplest case, a key typed event is produced by a single
key press (e.g., 'a'). Often, however, characters are produced by
series of key presses (e.g., 'shift' + 'a'), and the mapping from key
pressed events to key typed events may be many-to-one or
many-to-many. Key releases are not usually necessary to generate a key
typed event, but there are some cases where the key typed event is not
generated until a key is released (e.g., entering ASCII sequences via
the Alt-Numpad method in Windows). No key typed events are generated
for keys that don't generate characters (e.g., action keys, modifier
```

keys, etc.). The getKeyChar method always returns a valid Unicode character or CHAR_UNDEFINED. For key pressed and key released events, the getKeyCode method returns the event's keyCode. For key typed events, the getKeyCode method always returns VK_UNDEFINED.

The methods available are as follows:

```
public void setSource(Object newSource)

       Set the source of this KeyEvent. Dispatching this event
       subsequent to this operation will send this event to the new
       Object.
       Parameters:
              newSource - the KeyEvent's new source.

public int getKeyCode()

       Returns the integer key-code associated with the key in this
       event.
       Returns:
     the integer code for an actual key on the keyboard. (For
             KEY_TYPED events, keyCode is VK_UNDEFINED.)

public void setKeyCode(int keyCode)

       Set the keyCode value to indicate a physical key.
       Parameters:
              keyCode - an integer corresponding to an actual key
              on the keyboard.

public void setKeyChar(char keyChar)

       Set the keyChar value to indicate a logical character.
       Parameters:
              keyChar - a char corresponding to to the
              combination of keystrokes that make up this event.

public void setModifiers(int modifiers)

       Set the modifiers to indicate additional keys that were held
       down (shift, ctrl, alt, meta) defined as part of InputEvent.
       NOTE: use of this method is not recommended, because many AWT
       implementations do not recognize modifier changes. This is
       especially true for KEY_TYPED events where the shift modifier
       is changed.
       Parameters:
              modifiers - an integer combination of the modifier constants.
       See Also:
              InputEvent
```

```
public char getKeyChar()

       Returns the character associated with the key in this
       event. For example, the key-typed event for shift + "a" returns
       the value for "A".
       Returns:
              the Unicode character defined for this key event. If
              no valid Unicode character exists for this key event,
              keyChar is CHAR_UNDEFINED.

public static String getKeyText(int keyCode)

       Returns a String describing the keyCode, such as "HOME", "F1"
       or "A". These strings can be localized by changing the
       awt.properties file.
       Returns:
                string a text description for a
                physical key, identified by its keyCode

public static String getKeyModifiersText(int modifiers)

       Returns a String describing the modifier key(s), such as
       "Shift", or "Ctrl+Shift". These strings can be localized by
       changing the awt.properties file.
       Returns:
              string a text description of the combination of modifier
              keys that were held down during the event

public boolean isActionKey()

       Returns whether or not the key in this event is an "action"
       key, as defined in Event.java.
       Returns:
              boolean value, true if the key is an "action" key
       See Also:
              Event

public String paramString()

       Returns a parameter string identifying this event. This method
       is useful for event-logging and for debugging.
       Overrides:
              paramString in class ComponentEvent
       Returns:
              a string identifying the event and its attributes
```

5 Making Your Own Components and Graphics

Up to now, we have seen many AWT components Java provides to enable graphical user interfaces. What we have not seen how to do is to design our own components, and to be able to draw custom graphics. And, you will all need to use that in your projects. So, finally, we will start learning about custom graphics now.

In order to draw on, I think it is a simple idea to have a *blank* component, which serves no other purpose. Java provides exactly that, a `Canvas` class, which is a `Component`, but it has no function whatsoever. So, we shall first see the details of the `Canvas` class.

6 The `Canvas` Class

The Java documentation provides the following general information about the `Canvas` class:

```
A Canvas component represents a blank rectangular area of the screen
onto which the application can draw or from which the application can
trap input events from the user.

An application must subclass the Canvas class in order to get useful
functionality such as creating a custom component. The paint method
must be overridden in order to perform custom graphics on the canvas.
```

The `Canvas` class has only one method worth mentioning, and that is `void paint(Graphics g)`. Apart from that, it inherits all methods from `Component`.

The short story of how a canvas gets painted is as follows:

1. The system decides the component needs repainting.
2. It calls the `update()` method of the component.
3. By default (unless overriden), `update()` clears the component, and then calls the `paint()` method of the component.
4. The `paint()` method completely re-draws this component.

Both the `update()` and `paint()` methods receive an argument of type `Graphics`. It is the class that is used to actually draw things. It is time we saw the details of this class.

7 The `Graphics` Class

The Java documentation provides the following information about the graphics class:

```
The  Graphics  class is  the  abstract  base  class for  all
graphics  contexts that  allow an  application to  draw onto
components that are realized  on various devices, as well as
onto off-screen images.
```

A Graphics object encapsulates state information needed for the basic rendering operations that Java supports. This state information includes the following properties:

The Component object on which to draw.
A translation origin for rendering and clipping coordinates.
The current clip.
The current color.
The current font.
The current logical pixel operation function (XOR or Paint).
The current XOR alternation color (see setXORMode(java.awt.Color)).

Coordinates are infinitely thin and lie between the pixels of the output device. Operations that draw the outline of a figure operate by traversing an infinitely thin path between pixels with a pixel-sized pen that hangs down and to the right of the anchor point on the path. Operations that fill a figure operate by filling the interior of that infinitely thin path. Operations that render horizontal text render the ascending portion of character glyphs entirely above the baseline coordinate.

The graphics pen hangs down and to the right from the path it traverses. This has the following implications:

If you draw a figure that covers a given rectangle, that figure occupies one extra row of pixels on the right and bottom edges as compared to filling a figure that is bounded by that same rectangle.

If you draw a horizontal line along the same y coordinate as the baseline of a line of text, that line is drawn entirely below the text, except for any descenders.

All coordinates that appear as arguments to the methods of this Graphics object are considered relative to the translation origin of this Graphics object prior to the invocation of the method.

All rendering operations modify only pixels which lie within the area bounded by the current clip, which is specified by a Shape in user space and is controlled by the program using the Graphics object. This user clip is transformed into device space and combined with the device clip, which is defined by the visibility of windows and device extents. The combination of the user clip and device clip defines the composite clip, which determines the final clipping region. The user clip cannot be modified by the rendering system to reflect the resulting composite clip. The user clip can only be changed through the setClip or clipRect methods. All drawing or writing is done in the current color, using the current paint mode, and in the current font.

Here is a list of "interesting" methods of the `Graphics` class:

```
public abstract void setColor(Color c)
```

Sets this graphics context's current color to the specified color. All subsequent graphics operations using this graphics context use this specified color.

```
public abstract void drawLine(int x1, int y1, int x2, int y2)
```

Draws a line, using the current color, between the points (x1, y1) and (x2, y2) in this graphics context's coordinate system.

Parameters:
x1 – the first point's x coordinate.
y1 – the first point's y coordinate.
x2 – the second point's x coordinate.
y2 – the second point's y coordinate.

```
public void drawRect(int x, int y, int width, int height)
```

Draws the outline of the specified rectangle. The left and right edges of the rectangle are at x and x + width. The top and bottom edges are at y and y + height. The rectangle is drawn using the graphics context's current color.

```
public abstract void drawOval(int x, int y, int width,
                              int height)
```

Draws the outline of an oval. The result is a circle or ellipse that fits within the rectangle specified by the x, y, width, and height arguments.

The oval covers an area that is width + 1 pixels wide and height + 1 pixels tall..!

```
public abstract void fillRect(int x, int y, int width,
                              int height)
```

Fills the specified rectangle. The left and right edges of the rectangle are at x and x + width - 1. The top and bottom edges are at y and y + height - 1. The resulting rectangle

```
covers an area width pixels  wide by height pixels tall. The
rectangle  is filled  using the  graphics  context's current
color.

public abstract void fillOval(int x, int y, int width,
                              int height)

Fills an  oval bounded by  the specified rectangle  with the
current color.
```

8 An Example

As usual, you are probably pretty confused at this point. In order to reduce the amount of confusion, here is an example:

This is a class which extends `Canvas`. Type it in.

```
import java.awt.*;

public class ColorCanvas extends java.awt.Canvas {

    public void paint(Graphics g) {
        int h = getHeight();
        int w = getWidth();
        int value;

        for (int x=0; x<w; x++) {

            value = (int)((float)x / (float) w * 255.0);

            g.setColor(new Color(value, value, 0));
            g.drawLine(x, 0, x, h - 1);

        }
    }
}
```

First, try to figure out what it does. Next, test it in a `Frame`. You should instantiate a regular `Frame`, create a `ColorCanvas`, and add it to the `Frame`. When run, resize the `Frame` to see what happens. Does anything looking bad happen? Why does it happen?

9 Exercise

Write a new class which extends `Canvas`, and displays a 3x3 grid with (almost) equal-size cells inside it. It should keep working properly as the comonent is resized.