# SESSION 7

# Programming Languages for Objects

## Advanced GUI Programming

IT'S POSSIBLE TO PROGRAM a wide variety of GUI applications using only the techniques covered in SESSION 6. In many cases, the basic events, components, layouts, and graphics routines covered in that chapter suffice. But the Swing graphical user interface library is far richer than what we have seen so far, and it can be used to build highly sophisticated applications. This chapter is a further introduction to Swing and other aspects of GUI programming. Although the title of the chapter is "Advanced GUI Programming," it is still just an introduction

## Images and Resources

WE HAVE SEEN HOW TO USE the *Graphics* class to draw on a GUI component that is visible on the computer's screen. Often, however, it is useful to be able to create a drawing **off-screen**, in the computer's memory. It is also important to be able to work with images that are stored in files.

To a computer, an image is just a set of numbers. The numbers specify the color of each pixel in the image. The numbers that represent the image on the computer's screen are stored in a part of memory called a frame buffer. Many times each second, the computer's video card reads the data in the frame buffer and colors each pixel on the screen according to that data. Whenever the computer needs to make some change to the screen, it writes some new numbers to the frame buffer, and the change appears on the screen a fraction of a second later, the next time the screen is redrawn by the video card.

Since it's just a set of numbers, the data for an image doesn't have to be stored in a frame buffer. It can be stored elsewhere in the computer's memory. It can be stored in a file on the computer's hard disk. Just like any other data file, an image file can be downloaded over the Internet. Java includes standard classes and subroutines that can be used to copy image data from one part of memory to another and to get data from an image file and use it to display the image on the screen.

---

### 13.1.1 Images and Buffered Images

The class `java.awt.Image` represents an image stored in the computer's memory. There are two fundamentally different types of *Image*. One kind represents an image read from a source outside the program, such as from a file on the computer's hard disk or over a network connection. The second type is an image created by the program, by drawing to it using a graphics context. I refer to this second type as an off-screen canvas. An off-screen canvas is a region of the computer's memory that can be used as a drawing surface. It is possible to draw to an off-screen image using the same *Graphics* class that is used for drawing on the screen.

An *Image* of either type can be copied onto the screen (or onto an off-screen canvas) using methods that are defined in the *Graphics* class. This is most commonly done in the `paintComponent()` method of a *JComponent*. Suppose that `g` is the *Graphics* object that is a parameter to the `paintComponent()` method, and that `img` is of type *Image*. Then the statement

```
        g.drawImage(img, x, y, this);
```

will draw the image `img` in a rectangular area in the component. The integer-valued parameters `x` and `y` give the position of the upper-left corner of the rectangle in which the image is displayed, and the rectangle is just large enough to hold the image. The fourth parameter, `this` is there for technical reasons. In all cases in this book it will be either `this` or `null`. The parameter is of type *ImageObserver* and a non-null value is needed only when the complete image might not be available when the `drawImage()` method is called. This can happen, for example, if the image is being read from a file or downloaded over the network. You don't need it for an image that the program has created itself in the computer's memory or for an image that you are sure has already been completely loaded. In those cases, the image observer parameter can be `null`. However, even in those cases, using a non-null value does not cause any problems.

(In cases where you do need a non-null image observer, the special variable `this`, from Subsection 5.6.1, is usually appropriate. The image observer parameter is there for technical reasons having to do with the funny way Java treats image files. For most applications, you don't need to understand this, but here is how it works: `g.drawImage()` does not actually draw the image in all cases. In some circumstances, it is possible that the complete image is not available when this method is called. In that case, `g.drawImage()` merely **initiates** the drawing of the image and returns immediately. Pieces of the image are drawn later, asynchronously, as they become available. The question is, **how** do they get drawn? That's where the image observer comes in. When a piece of the image becomes available to be drawn, the system will inform the *ImageObserver*, and it is the responsibility of the observer to make sure that the new piece of the image will appear on the screen. Any *JComponent* object can act as an *ImageObserver*; it will call its `repaint()` method when notified that more of the image is available. The `drawImage` method returns a boolean value to indicate whether the image has actually been drawn or not when the method returns.)

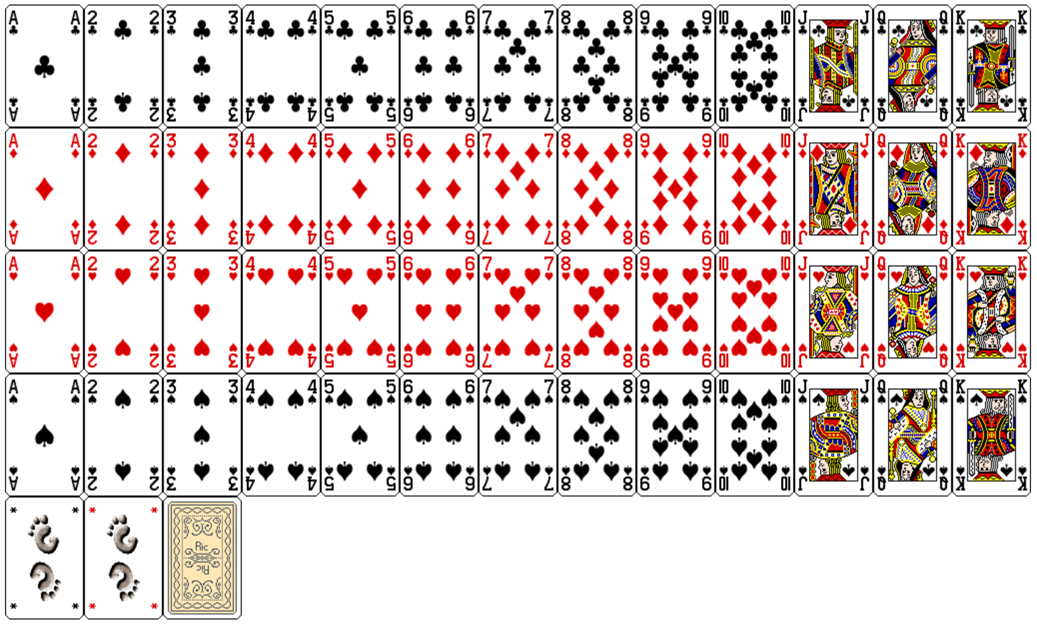
There are a few useful variations of the `drawImage()` method. For example, it is possible to scale the image as it is drawn to a specified width and height. This is done with the command

```
g.drawImage(img, x, y, width, height, imageObserver);
```

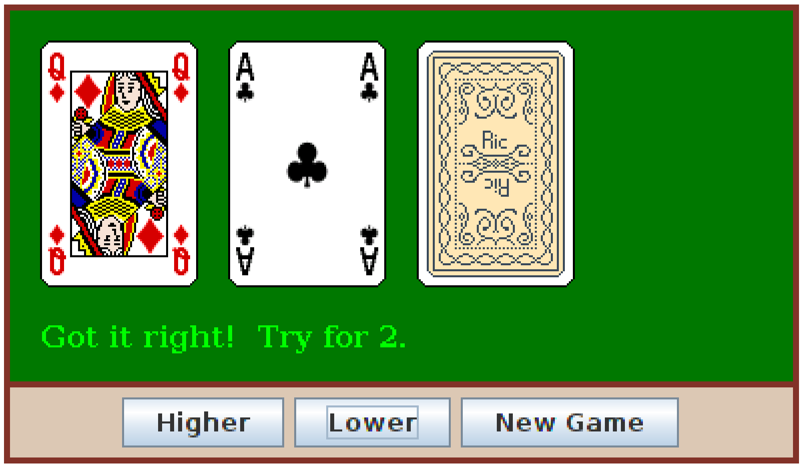
The parameters `width` and `height` give the size of the rectangle in which the image is displayed. Another version makes it possible to draw just part of the image. In the command:

```
g.drawImage(img, dest_x1, dest_y1, dest_x2, dest_y2,
                 source_x1, source_y1, source_x2, source_y2,
imageObserver);
```

the integers `source_x1`, `source_y1`, `source_x2`, and `source_y2` specify the top-left and bottom-right corners of a rectangular region in the source image. The integers `dest_x1`, `dest_y1`, `dest_x2`, and `dest_y2` specify opposite corners of a region in the destination graphics context. The specified rectangle in the image is drawn, with scaling if necessary, to the specified rectangle in the graphics context. For an example in which this is useful, consider a card game that needs to display 52 different cards. Dealing with 52 image files can be cumbersome and inefficient, especially for downloading over the Internet. So, all the cards might be put into a single image:



(This image is from the Gnome desktop project, http://www.gnome.org, and is shown here much smaller than its actual size.) Now just one *Image* object is needed. Drawing one card means drawing a rectangular region from the image. This technique is used in a variation of the sample program *HighLowGUI.java* from Subsection 6.6.6. In the original version, the cards are represented by textual descriptions such as "King of Hearts." In the new version, *HighLowWithImages.java*, the cards are shown as images. You should try the program, but here's how it looks with the card images:



In the program, the pictures of the cards are drawn using the following method. The instance variable `cardImages` is a variable of type *Image* that represents the image that is shown above, containing 52 cards, plus two Jokers and a face-down card. Each card is 79 by 123 pixels. These numbers are used, together with the suit and value of the card, to compute the corners of the source rectangle for the `drawImage()` command:

```
/**
 * Draws a card in a 79x123 pixel picture of a card with its
 * upper left corner at a specified point (x,y).  Drawing the card
 * requires the image file "cards.png".
 * @param g The graphics context used for drawing the card.
 * @param card The card that is to be drawn.  If the value is null,
then a
 * face-down card is drawn.
 * @param x the x-coord of the upper left corner of the card
 * @param y the y-coord of the upper left corner of the card
 */
public void drawCard(Graphics g, Card card, int x, int y) {
   int cx;    // x-coord of upper left corner of the card inside
cardsImage
   int cy;    // y-coord of upper left corner of the card inside
cardsImage
   if (card == null) {
      cy = 4*123;   // coords for a face-down card.
      cx = 2*79;
   }
   else {
      cx = (card.getValue()-1)*79;
      switch (card.getSuit()) {
      case Card.CLUBS:
         cy = 0;
         break;
      case Card.DIAMONDS:
```

```
                  cy = 123;
                  break;
               case Card.HEARTS:
                  cy = 2*123;
                  break;
               default:  // spades
                  cy = 3*123;
                  break;
               }
            }
            g.drawImage(cardImages,x,y,x+79,y+123,cx,cy,cx+79,cy+123,this);
         }
```

I will tell you later in this section how the image file, `cards.png`, can be loaded into the program.

---

In addition to images loaded from files, it is possible to create images by drawing to an off-screen canvas. An off-screen canvas can be represented by an object belonging to the class *BufferedImage*, which is defined in the package `java.awt.image`. *BufferedImage* is a subclass of *Image*, so that once you have a *BufferedImage*, you can copy it into a graphics context `g` using one of the `g.drawImage()` methods, just as you would do with any other image. A *BufferedImage* can be created using the constructor

```
public BufferedImage(int width, int height, int imageType)
```

where `width` and `height` specify the width and height of the image in pixels, and `imageType` can be one of several constants that are defined in the *BufferedImage*. The image type specifies how the color of each pixel is represented. One likely value for `imageType` is `BufferedImage.TYPE_INT_RGB`, which specifies that the color of each pixel is a usual RGB color, with red, green and blue components in the range 0 to 255. The image type `BufferedImage.TYPE_INT_ARGB` represents an RGB image with "transparency." The image type `BufferedImage.TYPE_BYTE_GRAY` can be used to create a grayscale image in which the only possible colors are shades of gray.

To draw to a *BufferedImage*, you need a graphics context that is set up to do its drawing on the image. If `OSC` is of type *BufferedImage*, then the method

```
OSC.createGraphics()
```

returns an object of type *Graphics2D* that can be used for drawing on the image. (*Graphics2D* is a subclass of *Graphics*. See Subsection 6.2.5 and the next section. This means you can also use the object as a regular *Graphics* drawing context.)

There are several reasons why a programmer might want to draw to an off-screen canvas. One is to simply keep a copy of an image that is shown on the screen. Remember that a picture that is drawn on a component can be lost, for example when the component is resized. This means that you have to be able to redraw the picture on demand, and that in turn means keeping enough information around to enable you to redraw the picture. One way to do this is to keep a copy of the picture in an off-screen canvas. Whenever the on-screen picture needs to be redrawn, you just have to copy the contents of the off-screen canvas onto the screen. Essentially, the off-screen canvas allows you to save a copy of the color of every individual pixel in the picture. The sample program *PaintWithOffScreenCanvas.java* is a little painting program that uses an off-screen canvas in this way. In this program, the user can draw curves, lines, and various shapes; a "Tool" menu allows the user to select the thing to be drawn. There is also an "Erase" tool and a "Smudge" tool that I will get to later. A *BufferedImage* is used to store the user's picture. When the user changes the picture, the changes are made to the image, and the changed image is then copied to the screen. No record is kept of the shapes that the user draws; the only record is the color of the individual pixels in the off-screen image. (You should contrast this with the program *SimplePaint2.java* in Subsection 7.3.3, where the user's drawing is recorded as a list of objects that represent the shapes that the user drew.)

You should try the program. Try drawing a Filled Rectangle on top of some other shapes. As you drag the mouse, the rectangle stretches from the starting point of the mouse drag to the current mouse location. As the mouse moves, the underlying picture seems to be unaffected -- parts of the picture can be covered up by the rectangle and later uncovered as the mouse moves, and *they are still there*. What this means is that the rectangle that is shown as you drag the mouse can't actually be part of the off-screen canvas, since drawing something into an image means changing the color of some pixels in the image. The previous colors of those pixels are not stored anywhere else and so are permanently lost. In fact, as you draw a line, rectangle, or oval in `PaintWithOffScreenCanvas`, the shape that is shown as you drag the mouse is not drawn to the off-screen canvas at all. Instead, the `paintComponent()` method draws the shape on top of the contents of the canvas. Only when you release the mouse does the shape become a permanent part of the off-screen canvas. This illustrates the point that when an off-screen canvas is used, not everything that is visible on the screen has to be drawn on the canvas. Some extra stuff can be drawn on top of the contents of the canvas by the `paintComponent()` method. The other tools are handled differently from the shape tools. For the curve, erase, and smudge tools, the changes are made to the canvas immediately, as the mouse is being dragged.

Let's look at how an off-screen canvas is used in this program. The canvas is represented by an instance variable, `OSC`, of type *BufferedImage*. The size of the canvas must be the same size as the panel on which the canvas is displayed. The size can be determined by calling the `getWidth()` and `getHeight()` instance methods of the panel. Furthermore, when the canvas is first created, it should be filled with the background color, which is represented in the program by an instance variable named `fillColor`. All this is done by the method:

```
/**
 * This method creates the off-screen canvas and fills it with the
current
 * fill color.
 */
private void createOSC() {
   OSC = new
BufferedImage(getWidth(),getHeight(),BufferedImage.TYPE_INT_RGB);
   Graphics osg = OSC.createGraphics();
```

```
      osg.setColor(fillColor);
      osg.fillRect(0,0,getWidth(),getHeight());
      osg.dispose();
   }
```

Note how it uses `OSC.createGraphics()` to obtain a graphics context for drawing to the image. Also note that the graphics context is disposed at the end of the method. When you create a graphics context, it is good practice to dispose of it when you are finished with it. There still remains the problem of where to call this method. The problem is that the width and height of the panel object are not set until some time after the panel object is constructed. If `createOSC()` is called in the constructor, `getWidth()` and `getHeight()` will return the value zero and we won't get an off-screen image of the correct size. The approach that I take in `PaintWithOffScreenCanvas` is to call `createOSC()` in the `paintComponent()` method, the first time the `paintComponent()` method is called. At that time, the size of the panel has definitely been set, but the user has not yet had a chance to draw anything. With this in mind you are ready to understand the `paintComponent()` method:

```
   public void paintComponent(Graphics g) {

      /* First create the off-screen canvas, if it does not already
   exist. */

      if (OSC == null)
         createOSC();

      /* Copy the off-screen canvas to the panel.  Since we know that
   the
         image is already completely available, the fourth,
   "ImageObserver"
         parameter to g.drawImage() can be null.  Since the canvas
   completely
         fills the panel, there is no need to call
   super.paintComponent(g). */

      g.drawImage(OSC,0,0,null);

      /* If the user is currently dragging the mouse to draw a line,
   oval,
         or rectangle, draw the shape on top of the image from the
   off-screen
         canvas, using the current drawing color.  (This is not done
   if the
         user is drawing a curve or using the smudge, curve, or erase
   tool.) */

      if (dragging && SHAPE_TOOLS.contains(currentTool)) {
         g.setColor(currentColor);
         putCurrentShape(g);
      }

   }
```

Here, dragging is a boolean instance variable that is set to true while the user is dragging the mouse, and currentTool tells which tool is currently in use. The possible tools are defined by an enum named *Tool*, and SHAPE_TOOLS is a variable of type *EnumSet<Tool>* that contains the line, oval, rectangle, filled oval, and filled rectangle tools. (See Subsection 10.2.4.)

You might notice that there is a problem if the size of the panel is ever changed, since the size of the off-screen canvas will not be changed to match. The PaintWithOffScreenCanvas program does not allow the user to resize the program's window, so this is not an issue in that program. If we want to allow resizing, however, a new off-screen canvas must be created whenever the size of the panel changes. One simple way to do this is to check the size of the canvas in the paintComponent() method and to create a new canvas if the size of the canvas does not match the size of the panel:

```
if (OSC == null || getWidth() != OSC.getWidth() || getHeight() != 
OSC.getHeight())
   createOSC();
```

Of course, this will discard the picture that was contained in the old canvas unless some arrangement is made to copy the picture from the old canvas to the new one before the old canvas is discarded.

The other point in the program where the off-screen canvas is used is during a mouse-drag operation, which is handled in the mousePressed(), mouseDragged(), and mouseReleased() methods. The strategy that is implemented was discussed above. Shapes are drawn to the off-screen canvas only at the end of the drag operation, in the mouseReleased() method. However, as the user drags the mouse, the part of the image over which the shape appears is re-copied from the canvas onto the screen each time the mouse is moved. Then the paintComponent() method draws the shape that the user is creating on top of the image from the canvas. For the non-shape (curve, smudge, and erase) tools, on the other hand, changes are made directly to the canvas, and the region that was changed is repainted so that the change will appear on the screen. (By the way, the program uses a version of the repaint() method that repaints just a part of a component. The command repaint(x,y,width,height) tells the system to repaint the rectangle with upper left corner (x,y) and with the specified width and height. This can be substantially faster than repainting the entire component.) See the source code, *PaintWithOffScreenCanvas.java*, if you want to see how it's all done.

---

One traditional use of off-screen canvasses is for double buffering. In double-buffering, the off-screen image is an exact copy of the image that appears on screen; whenever the on-screen picture needs to be redrawn, the new picture is drawn step-by-step to an off-screen image. This can take some time. If all this drawing were done on screen, the user might see the image flicker as it is drawn. Instead, the long drawing process takes place off-screen and the completed image is then copied very quickly onto the screen. The user doesn't see all the steps involved in redrawing. This technique can be used to implement smooth, flicker-free animation.

The term "double buffering" comes from the term "frame buffer," which refers to the region in memory that holds the image on the screen. In fact, true double buffering uses two frame buffers. The video card can display either frame buffer on the screen and can switch instantaneously from one frame buffer to the other. One frame buffer is used to draw a new image for the screen. Then the video card is told to switch from one frame buffer to the other. No copying of memory is involved. Double-buffering as it is implemented in Java does require copying, which takes some time and is not perfectly flicker-free.

In Java's older AWT graphical API, it was up to the programmer to do double buffering by hand. In the Swing graphical API, double buffering is applied automatically by the system, and the programmer doesn't have to worry about it. (It is possible to turn this automatic double buffering off in Swing, but there is seldom a good reason to do so.)

One final historical note about off-screen canvasses: There is an alternative way to create them. The *Component* class defines the following instance method, which can be used in any GUI component object:

```
public Image createImage(int width, int height)
```

This method creates an *Image* with a specified width and height. You can use this image as an off-screen canvas in the same way that you would a *BufferedImage*. In fact, you can expect that in a modern version of Java, the image that is returned by this method is in fact a *BufferedImage*. The `createImage()` method was part of Java from the beginning, before the *BufferedImage* class was introduced.

---

### 13.1.2 Working With Pixels

One good reason to use a *BufferedImage* is that it allows easy access to the colors of individual pixels. If `image` is of type *BufferedImage*, then we have the methods:

- `image.getRGB(x,y)` -- returns an int that encodes the color of the pixel at coordinates `(x,y)` in the image. The values of the integers `x` and `y` must lie within the image. That is, it must be true that `0 <= x < image.getWidth()` and `0 <= y < image.getHeight()`; if not, then an exception is thrown.
- `image.setRGB(x,y,rgb)` -- sets the color of the pixel at coordinates `(x,y)` to the color encoded by `rgb`. Again, `x` and `y` must be in the valid range. The third parameter, `rgb`, is an integer that encodes the color.

These methods use integer codes for colors. If `c` is of type *Color*, the integer code for the color can be obtained by calling `c.getRGB()`. Conversely, if `rgb` is an integer that encodes a color, the corresponding *Color* object can be obtained with the constructor call `new Color(rgb)`. This means that you can use

```
Color c = new Color( image.getRGB(x,y) )
```

to get the color of a pixel as a value of type *Color*. And if `c` is of type *Color*, you can set a pixel to that color with

```
        image.setRGB( x, y, c.getRGB() );
```

The red, green, and blue components of a color are represented as 8-bit integers, in the range 0 to 255. When a color is encoded as a single int, the blue component is contained in the eight low-order bits of the int, the green component in the next lowest eight bits, and the red component in the next eight bits. (The eight high order bits store the "alpha component" of the color, which we'll encounter in the next section.) It is easy to translate between the two representations using the shift operators << and >> and the bitwise logical operators `&` and `|`. (I have not covered these operators previously in this book. Briefly: If `A` and `B` are integers, then `A << B` is the integer obtained by shifting each bit of `A`, `B` bit positions to the left; `A >> B` is the integer obtained by shifting each bit of `A`, `B` bit positions to the right; `A & B` is the integer obtained by applying the logical **and** operation to each pair of bits in `A` and `B`; and `A | B` is obtained similarly, using the logical **or** operation. For example, using 8-bit binary numbers, we have: `01100101 & 10100001` is `00100001`, while `01100101 | 10100001` is `11100101`.) You don't necessarily need to understand these operators. Here are incantations that you can use to work with color codes:

```
        /* Suppose that rgb is an int that encodes a color.
           To get separate red, green, and blue color components: */

        int red = (rgb >> 16) & 0xFF;
        int green = (rgb >> 8) & 0xFF;
        int blue = rgb & 0xFF;

        /* Suppose that red, green, and blue are color components in
           the range 0 to 255.  To combine them into a single int: */

        int rgb = (red << 16) | (green << 8) | blue;
```

---

An example of using pixel colors in a *BufferedImage* is provided by the smudge tool in the sample program *PaintWithOffScreenCanvas.java*. The purpose of this tool is to smear the colors of an image, as if it were drawn in wet paint. For example, if you rub the middle of a black rectangle with the smudge tool, you'll get something like this:

This is an effect that can only be achieved by manipulating the colors of individual pixels! Here's how it works: when the user presses the mouse using the smudge tool, the color components of a 7-by-7 block of pixels are copied from the off-screen canvas into arrays named `smudgeRed`, `smudgeGreen` and `smudgeBlue`. This is done in the `mousePressed()` routine with the following code:

```
int w = OSC.getWidth();
int h = OSC.getHeight();
int x = evt.getX();
int y = evt.getY();
for (int i = 0; i < 7; i++)
   for (int j = 0; j < 7; j++) {
      int r = y + j - 3;
      int c = x + i - 3;
      if (r < 0 || r >= h || c < 0 || c >= w) {
            // A -1 in the smudgeRed array indicates that the
            // corresponding pixel was outside the canvas.
         smudgeRed[i][j] = -1;
      }
      else {
         int color = OSC.getRGB(c,r);
         smudgeRed[i][j] = (color >> 16) & 0xFF;
         smudgeGreen[i][j] = (color >> 8) & 0xFF;
         smudgeBlue[i][j] = color & 0xFF;
      }
   }
```

The arrays are of type `double[][]` because I am going to do some computations with them that require real numbers. As the user moves the mouse, the colors in the array are blended with the colors in the image, just as if you were mixing wet paint by smudging it with your finger. That is, the colors at the new mouse position in the image are replaced with a weighted average of the current colors in the image and the colors in the arrays. This has the effect of moving some of the color from the previous mouse position to the new mouse position. At the same time, the colors in the arrays are replaced by a weighted average of the old colors in the arrays and the colors from the image. This has the effect of moving some color from the image into the arrays. This is done using the following code for each pixel position, `(c,r)`, in a 7-by-7 block around the new mouse location:

```
int curCol = OSC.getRGB(c,r);
int curRed = (curCol >> 16) & 0xFF;
int curGreen = (curCol >> 8) & 0xFF;
int curBlue = curCol & 0xFF;
int newRed = (int)(curRed*0.7 + smudgeRed[i][j]*0.3);
int newGreen = (int)(curGreen*0.7 + smudgeGreen[i][j]*0.3);
int newBlue = (int)(curBlue*0.7 + smudgeBlue[i][j]*0.3);
int newCol = newRed << 16 | newGreen << 8 | newBlue;
OSC.setRGB(c,r,newCol);
smudgeRed[i][j] = curRed*0.3 + smudgeRed[i][j]*0.7;
smudgeGreen[i][j] = curGreen*0.3 + smudgeGreen[i][j]*0.7;
smudgeBlue[i][j] = curBlue*0.3 + smudgeBlue[i][j]*0.7;
```

### 13.1.3 Resources

Throughout this textbook, up until now, we have been thinking of a program as made up entirely of Java code. However, programs often use other types of data, including images, sounds, and text, as part of their basic structure. These data are referred to as resources. An example is the image file, `cards.png`, that was used in the *HighLowWithImages.java* program earlier in this section. This file is part of the program. The program needs it in order to run. The user of the program doesn't need to know that this file exists or where it is located; as far as the user is concerned, it is just part of the program. The program of course, does need some way of locating the resource file and loading its data.

Resources are ordinarily stored in files that are in the same locations as the compiled class files for the program. Class files are located and loaded by something called a class loader, which is represented in Java by an object of type *ClassLoader*. A class loader has a list of locations where it will look for class files. This list is called the class path. It includes the location where Java's standard classes are stored. It generally includes the current directory. If the program is stored in a jar file, the jar file is included on the class path. In addition to class files, a *ClassLoader* is capable of finding resource files that are located on the class path or in subdirectories of locations that are on the class path.

The first step in using a resource is to obtain a *ClassLoader* and to use it to locate the resource file. In the `HighLowWithImages` program, this is done with:

```
ClassLoader cl = getClass().getClassLoader();
URL imageURL = cl.getResource("cards.png");
```

The idea of the first line is that in order to get a class loader, you have to ask a class that was loaded by the class loader. Here, `getClass()` is a is a reference to object that represents the actual class, *HighLowWithImages*. The `getClass()` method is an instance method in class *Object* and so can be used with any object. Another way to get a reference to a class loader is to use `ClassName.class`, where *ClassName* is the name of any class. For example, I could have used `HighLoadWithImages.class.getClassLoader()` to get the class loader in this case.

The second line in the above code uses the class loader to locate the resource file named `cards.png`. The return value of `cl.getResource()` is of type `java.net.URL`, and it represents the location of the resource rather than the resource itself. If the resource file cannot be found, then the return value is null. The class *URL* was discussed in Subsection 11.4.1.

Often, resources are stored not directly on the class path but in a subdirectory. In that case, the parameter to `getResource()` must be a path name that includes the directory path to the resource. For example, suppose that the image file "cards.png" were stored in a directory named `images` inside a directory named `resources`, where `resources` is directly on the class path. Then the path to the file is "resources/images/cards.png" and the command for locating the resource would be

```
URL imageURL = cl.getResource("resources/images/cards.png");
```

Once you have a *URL* that represents the location of a resource file, you could use a *URLConnection*, as discussed in Subsection 11.4.1, to read the contents of that file. However, Java provides more convenient methods for loading several types of resources. For loading image resources, a convenient method is available in the class `java.awt.Toolkit`. It can be used as in the following line from HighLowWithImages, where cardImages is an instance variable of type *Image* and imageURL is the *URL* that represents the location of the image file:

```
cardImages = Toolkit.getDefaultToolkit().createImage(imageURL);
```

This still does not load the image completely -- that will only be done later, for example when cardImages is used in a drawImage command. Another technique, which does read the image completely, is to use the ImageIO.read() method, which will be discussed below in Subsection 13.1.5

---

Sounds represent another kind of resource that a program might want to use. For some reason, the easiest way to playing a sound is to use a static method in the *Applet* class, in package java.awt:

```
public static AudioClip newAudioClip(URL soundURL)
```

Since this is a static method, it can be used in any program, not just in applets, simply by calling it as Applet.newAudioClip(soundURL). The parameter is the URL of a sound resource, and the return value is of type java.applet.AudioClip. Once you have an *AudioClip*, you can call its play() method to play the audio clip from the beginning.

Here is a method that puts all this together to load and play the sound from an audio resource file:

```
private void playAudioResource(String audioResourceName) {
   ClassLoader cl = getClass().getClassLoader();
   URL resourceURL = cl.getResource(audioResourceName);
   if (resourceURL != null) {
      AudioClip sound = Applet.newAudioClip(resourceURL);
      sound.play();
   }
}
```

This method is from a sample program *SoundAndCursorDemo.java* that will be discussed in the next subsection. Of course, if you plan to reuse the sound often, it would be better to load the sound once into an instance variable of type *AudioClip*, which could then be used to play the sound any number of times, without the need to reload it each time.

The *AudioClip* class supports audio files in the common WAV, AIFF, and AU formats.

---

### 13.1.4 Cursors and Icons

The position of the mouse is represented on the computer's screen by a small image called a cursor. In Java, the cursor is represented by an object of type `java.awt.Cursor`. A *Cursor* has an associated image. It also has a hot spot, which is a *Point* that specifies the pixel within the image that corresponds to the exact position on the screen where the mouse is pointing. For example, for a typical "arrow" cursor, the hot spot is the tip of the arrow. For a "crosshair" cursor, the hot spot is the center of the crosshairs.

The *Cursor* class defines several standard cursors, which are identified by constants such as `Cursor.CROSSHAIR_CURSOR` and `Cursor.DEFAULT_CURSOR`. You can get a standard cursor by calling the `static` method `Cursor.getPredefinedCursor(code)`, where `code` is one of the constants that identify the standard cursors. It is also possible to create a custom cursor from an *Image*. The *Image* might be obtained as an image resource, as described in the previous subsection. It could even be a *BufferedImage* that you create in your program. It should be small, maybe 16-by-16 or 24-by-24 pixels. (Some platforms might only be able to handle certain cursor sizes; see the documentation for `Toolkit.getBestCursorSize()` for more information.) A custom cursor can be created by calling the `static` method `createCustomCursor()` in the *Toolkit* class:

```
Cursor c =
Toolkit.getDefaultToolkit().createCustomCursor(image,hotSpot,name);
```

where `hotSpot` is of type *Point* and `name` is a *String* that will act as a name for the cursor (and which serves no real purpose that I know of).

Cursors are associated with GUI components. When the mouse moves over a component, the cursor changes to whatever *Cursor* is associated with that component. To associate a *Cursor* with a component, call the component's instance method `setCursor(cursor)`. For example, to set the cursor for a *JPanel*, `panel`, to be the standard "wait" cursor:

```
panel.setCursor( Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR) );
```

To reset the cursor to be the default cursor, you can use:

```
panel.setCursor( Cursor.getDefaultCursor() );
```

To set the cursor to be an image that is defined in an image resource file named `imageResource`, you might use:

```
ClassLoader cl = getClass().getClassLoader();
URL resourceURL = cl.getResource(imageResource);
if (resourceURL != null) {
   Toolkit toolkit = Toolkit.getDefaultToolkit();
   Image image = toolkit.createImage(resourceURL);
   Point hotSpot = new Point(7,7);
   Cursor cursor = toolkit.createCustomCursor(image, hotSpot,
"mycursor");
```

```
            panel.setCursor(cursor);
         }
```

The sample program *SoundAndCursorDemo.java* shows how to use predefined and custom cursors and how to play sounds from resource files. The program has several buttons that you can click. Some of the buttons change the cursor that is associated with the main panel of the program. Some of the buttons play sounds. When you play a sound, the cursor is reset to be the default cursor.

Another standard use of images in GUI interfaces is for icons. An icon is simply a small picture. As we'll see in Section 13.3, icons can be used on Java's buttons, menu items, and labels; in fact, for our purposes, an icon is simply an image that can be used in this way.

An icon is represented by an object of type *Icon*, which is actually an `interface` rather than a class. The class *ImageIcon*, which implements the *Icon* interface, is used to create icons from *Images*. If `image` is a (rather small) *Image*, then the constructor call `new ImageIcon(image)` creates an *ImageIcon* whose picture is the specified image. Often, the image comes from a resource file. We will see examples of this later in this chapter

### 13.1.5 Image File I/O

The class `javax.imageio.ImageIO` makes it easy to save images from a program into files and to read images from files into a program. This would be useful in a program such as `PaintWithOffScreenCanvas`, so that the users would be able to save their work and to open and edit existing images. (See Exercise 13.1.)

There are many ways that the data for an image could be stored in a file. Many standard formats have been created for doing this. Java supports at least three standard image formats: PNG, JPEG, and GIF. (Individual implementations of Java might support more.) The JPEG format is "lossy," which means that the picture that you get when you read a JPEG file is only an approximation of the picture that was saved. Some information in the picture has been lost. Allowing some information to be lost makes it possible to compress the image into a lot fewer bits than would otherwise be necessary. Usually, the approximation is quite good. It works best for photographic images and worst for simple line drawings. The PNG format, on the other hand is "lossless," meaning that the picture in the file is an exact duplicate of the picture that was saved. A PNG file is compressed, but not in a way that loses information. The compression works best for images made up mostly of large blocks of uniform color; it works **worst** for photographic images. GIF is an older format that is limited to just 256 colors in an image; it has mostly been superseded by PNG.

Suppose that `image` is a *BufferedImage*. The image can be saved to a file simply by calling

```
         ImageIO.write( image, format, file )
```

where `format` is a *String* that specifies the image format of the file and `file` is a *File* that specifies the file that is to be written. (See Subsection 11.2.2 for information about the *File* class.) The `format` string should ordinarily be either `"PNG"` or `"JPEG"`, although other formats might be supported.

`ImageIO.write()` is a `static` method in the *ImageIO* class. It returns a boolean value that is `false` if the image format is not supported. That is, if the specified image format is not supported, then the image is **not** saved, but no exception is thrown. This means that you should always check the return value! For example:

```
boolean hasFormat = ImageIO.write(OSC,format,selectedFile);
if ( ! hasFormat )
   throw new Exception(format + " format is not available.");
```

If the image format **is** recognized, it is still possible that an *IOException* might be thrown when the attempt is made to send the data to the file.

Usually, the file to be used in `ImageIO.write()` will be selected by the user using a *JFileChooser*, as discussed in Subsection 11.2.3. For example, here is a typical method for saving an image. (The use of "this" as a parameter in several places assumes that this method is defined in a subclass of *JComponent*.)

```
/**
 * Attempts to save an image to a file selected by the user.
 * @param image the BufferedImage to be saved to the file
 * @param format the format of the image, probably either "PNG" or
"JPEG"
 */
private void doSaveFile(BufferedImage image, String format) {
   if (fileDialog == null)
      fileDialog = new JFileChooser();
   fileDialog.setSelectedFile(new File("image." +
format.toLowerCase()));
   fileDialog.setDialogTitle("Select File For Saved Image");
   int option = fileDialog.showSaveDialog(this);
   if (option != JFileChooser.APPROVE_OPTION)
      return;  // User canceled or clicked the dialog's close box.
   File selectedFile = fileDialog.getSelectedFile();
   if (selectedFile.exists()) {  // Ask the user whether to replace
the file.
      int response = JOptionPane.showConfirmDialog( null,
            "The file \"" + selectedFile.getName()
            + "\" already exists.\nDo you want to replace it?",
            "Confirm Save",
            JOptionPane.YES_NO_OPTION,
            JOptionPane.WARNING_MESSAGE );
      if (response != JOptionPane.YES_OPTION)
         return;  // User does not want to replace the file.
   }
   try {
      boolean hasFormat = ImageIO.write(image,format,selectedFile);
      if ( ! hasFormat )
```

```
            throw new Exception(format + " format is not available.");
      }
      catch (Exception e) {
         JOptionPane.showMessageDialog(this,
                        "Sorry, an error occurred while trying to
save image."));
         e.printStackTrace();
      }
   }
```

---

The *ImageIO* class also has a `static read()` method for reading an image from a file into a program. The method

```
ImageIO.read( inputFile )
```

takes a variable of type *File* as a parameter and returns a *BufferedImage*. The return value is `null` if the file does not contain an image that is stored in a supported format. Again, no exception is thrown in this case, so you should always be careful to check the return value. It is also possible for an *IOException* to occur when the attempt is made to read the file. There is another version of the `read()` method that takes an *InputStream* instead of a file as its parameter, and a third version that takes a *URL*.

Earlier in this section, we encountered another method for reading an image from a *URL*, the `createImage()` method from the *Toolkit* class. The difference is that `ImageIO.read()` reads the image data completely and stores the result in a *BufferedImage*. On the other hand, `createImage()` does not actually read the data; it really just stores the image location and the data won't be read until later, when the image is used. This has the advantage that the `createImage()` method itself can complete very quickly. `ImageIO.read()`, on the other hand, can take some time to execute.

---

## Fancier Graphics

---

THE GRAPHICS COMMANDS provided by the *Graphics* class are sufficient for many purposes. However, recent versions of Java provide a much larger and richer graphical toolbox in the form of the class `java.awt.Graphics2D`. I mentioned *Graphics2D* in Subsection 6.2.5 and promised to discuss it further in this chapter. You have already seen a few of the ideas that are covered in this section, at least briefly, but I cover them in more detail here.

*Graphics2D* is a subclass of *Graphics*, so all of the graphics commands that you already know can be used with a *Graphics2D* object. In fact, when you obtain a *Graphics* context for drawing on a Swing component, the graphics object is actually of type *Graphics2D* and can be type-cast to gain access to the advanced *Graphics2D* graphics commands. Furthermore, *BufferedImage* has the instance method, `createGraphics()`, that returns a graphics context of type *Graphics2D*. As mentioned in Subsection 6.2.5, to use *Graphics2D* commands in the `paintComponent()` method of a Swing component, you can use code of the form:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics g2 = (Graphics2D)g;
      .
      .    // Draw to the component using g2 (and g).
      .
}
```

Note that when you do this, `g` and `g2` are just two variables that refer to the same object, so they both draw to the same drawing surface and have the same state. When properties of `g2`, such as drawing color, are changed, the changes also apply to `g`. By saying

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

you can obtain a newly created graphics context. The object created by `g.create()` is a graphics context that draws to the same drawing surface as `g` and that initially has all the same properties as `g`. However, it is a separate object, so that changing properties in `g2` has no effect on `g`. This can be useful if you want to keep an unmodified copy of the original graphics context around for some drawing operations. (In this case, it is good practice to call `g2.dispose()` to dispose of the new graphics context when you are finished using it.)

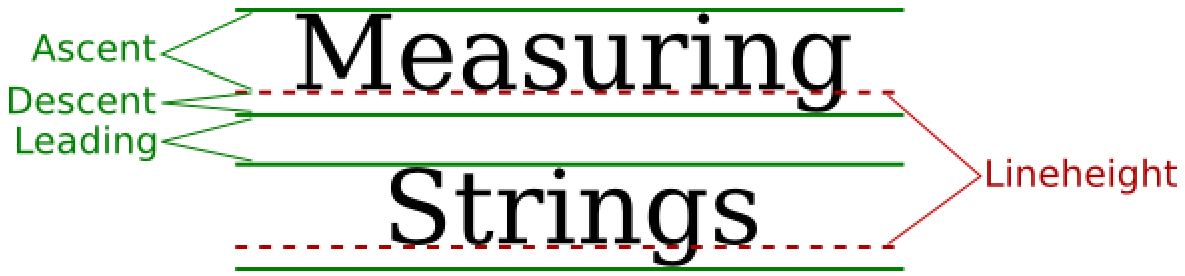
---

### 13.2.1 Measuring Text

Although this section is mostly about *Graphics2D*, we start with a topic that has nothing to do with it.

Often, when drawing a string, it's important to know how big the image of the string will be. For example, you need this information if you want to center a string in a component. Or if you want to know how much space to leave between two lines of text, when you draw them one above the other. Or if the user is typing the string and you want to position a cursor at the end of the string. In Java, questions about the size of a string can be answered by an object belonging to the standard class `java.awt.FontMetrics`.

There are several lengths associated with any given font. Some of them are shown in this illustration:



The dashed red lines in the illustration are the baselines of the two lines of text. The baseline of a string is the line on which the bases of the characters rest. The suggested distance between two baselines, for single-spaced text, is known as the lineheight of the font. The ascent is the distance that tall characters can rise above the baseline, and the descent is the distance that tails like the one on the letter "g" can descend below the baseline. The ascent and descent do not add up to the lineheight, because there should be some extra space between the tops of characters in one line and the tails of characters on the line above. The extra space is called leading. (The term comes from the time when lead blocks were used for printing. Characters were formed on blocks of lead that were lined up to make up the text of a page, covered with ink, and pressed onto paper to print the page. Extra, blank "leading" was used to separate the lines of characters.) All these quantities can be determined by calling instance methods in a *FontMetrics* object. There are also methods for determining the width of a character and the total width of a string of characters.

Recall that a font in Java is represented by the class *Font*. A *FontMetrics* object is associated with a given font and is used to measure characters and strings in that font. If `font` is of type *Font* and `g` is a graphics context, you can get a *FontMetrics* object for the font by calling `g.getFontMetrics(font)`. If `fm` is the variable that refers to the *FontMetrics* object, then the ascent, descent, leading, and lineheight of the font can be obtained by calling `fm.getAscent()`, `fm.getDescent()`, `fm.getLeading()`, and `fm.getHeight()`. If `ch` is a character, then `fm.charWidth(ch)` is the width of the character when it is drawn in that font. If `str` is a string, then `fm.stringWidth(str)` is the width of the string when drawn in that font. For example, here is a `paintComponent()` method that shows the message "Hello World" in the exact center of the component:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);

   int strWidth, strHeight; // Width and height of the string.
   int centerX, centerY;    // Coordinates of the center of the
component.
   int baseX, baseY;        // Coordinates of the baseline of the
string.
   int topOfString;         // y-coordinate of the top of the
string.

   centerX = getWidth() / 2;
   centerY = getHeight() / 2;

   Font font = g.getFont();  // What font will g draw in?
   FontMetrics fm = g.getFontMetrics(font);
   strWidth = fm.stringWidth("Hello World");
   strHeight = fm.getAscent();  // Note: There are no tails on and
                                //   of the chars in the string!
So we
                                //   don't need to account for
descent.

   baseX = centerX - (strWidth/2);  // Move back from center by
half the
                                    //    width of the string.

   topOfString = centerY - (strHeight/2);  // Move up from center
by half
                                           //   the height of the
string.

   baseY = topOfString + fm.getAscent();  // Baseline is
fm.getAscent() pixels
                                          //   below the top of the
string.

   g.drawString("Hello World", baseX, baseY); // Draw the string.
}
```

For the height of the string in this method, I use `fm.getAscent()`. If I were drawing "Goodbye World" instead of "Hello World," I would have used `fm.getAscent() + fm.getDescent()`, where the descent is added to the height in order to take into account the tail on the "y" in "Goodbye". The value of `baseX` is computed to be the amount of space between the left edge of the component and the start of the string. It is obtained by subtracting half the width of the string from the horizontal center of the component. This will center the string horizontally in the component. The next line computes the position of the top of the string in the same way. However, to draw the string, we need the y-coordinate of the baseline, not the y-coordinate of the top of the string. The baseline of the string is below the top of the string by an amount equal to the ascent of the font.

There is an example of centering a two-line block of text in the sample program *TransparencyDemo.java*, which is discussed in the next subsection.

---

### 13.2.2 Transparency

A color is represented by red, blue, and green components. In Java's usual representation, each component is an eight-bit number in the range 0 to 255. The three color components can be packed into a 32-bit integer, but that only accounts for 24 bits in the integer. What about the other eight bits? They don't have to be wasted. They can be used as a fourth component of the color, the alpha component. The alpha component can be used in several ways, but it is most commonly associated with transparency. When you draw with a transparent color, it's like laying down a sheet of colored glass. It doesn't completely obscure the part of the image that is colored over. Instead, the background image is blended with the transparent color that is used for drawing -- as if you were looking at the background through colored glass. This type of drawing is properly referred to as alpha blending, and it is not equivalent to true transparency; nevertheless, most people refer to it as transparency.

The value of the alpha component determines how transparent that color is. Actually, the alpha component gives the opaqueness of the color. Opaqueness is the opposite of transparency. If something is fully opaque, you can't see through it at all; if something is almost fully opaque, then it is just a little transparent; and so on. When the alpha component of a color has the maximum possible value, the color is fully opaque. When you draw with a fully opaque color, that color simply replaces the color of the background over which you draw. Except for a little example in Subsection 5.3.3, this is the only type of color that we have used up until now. If the alpha component of a color is zero, then the color is perfectly transparent, and drawing with that color has no effect at all. Intermediate values of the alpha component give partially opaque colors that will blend with the background when they are used for drawing.

The sample program *TransparencyDemo.java* can help you to understand transparency. When you run the program you will see a display area containing a triangle, an oval, a rectangle, and some text. Sliders at the bottom of the window allow you to control the degree of transparency of each shape. When a slider is moved all the way to the right, the corresponding shape is fully opaque; all the way to the left, and the shape is fully transparent.

---

Colors with alpha components were introduced in Java along with *Graphics2D*, but they can be used with ordinary *Graphics* objects as well. To specify the alpha component of a color, you can create the *Color* object using one of the following constructors from the *Color* class:

```
public Color(int red, int green, int blue, int alpha);

public Color(float red, float green, float blue, float alpha);
```

In the first constructor, all the parameters must be integers in the range 0 to 255. In the second, the parameters must be in the range 0.0 to 1.0. For example,

```
Color transparentRed = new Color( 255, 0, 0, 200 );
```

makes a slightly transparent red, while

```
Color tranparentCyan = new Color( 0.0F, 1.0F, 1.0F, 0.5F);
```

makes a blue-green color that is 50% opaque. (The advantage of the constructor that takes parameters of type float is that it lets you think in terms of percentages.) When you create an ordinary RGB color, as in `new Color(255,0,0)`, you just get a fully opaque color.

Once you have a transparent color, you can use it in the same way as any other color. That is, if you want to use a *Color* `c` to draw in a graphics context `g`, you just say `g.setColor(c)`, and subsequent drawing operations will use that color. As you can see, transparent colors are very easy to use.

---

A *BufferedImage* with image type `BufferedImage.TYPE_INT_ARGB` can use transparency. The color of each pixel in the image can have its own alpha component, which tells how transparent that pixel will be when the image is drawn over some background. A pixel whose alpha component is zero is perfectly transparent, and has no effect at all when the image is drawn; in effect, it's not part of the image at all. It is also possible for pixels to be partly transparent. When an image is saved to a file, information about transparency might be lost, depending on the file format. The PNG image format supports transparency; JPEG does not. (If you look at the images of playing cards that are used in the program `HighLowWithImages` in Subsection 13.1.1, you might notice that the tips of the corners of the cards are fully transparent. The card images are from a PNG file, *cards.png*.)

An ARGB *BufferedImage* should be fully transparent when it is first created, but if you want to make sure, here is one way of doing so: The *Graphics2D* class has a method `setBackground()` that can be used to set a background color for the graphics context, and it has a `clearRect()` method that fills a rectangle with the current background color. To create a fully transparent image with width `w` and height `h`, you can use:

```
BufferedImage image = new BufferedImage(w, h,
BufferedImage.TYPE_INT_ARGB);
Graphics2D g2 = image.createGraphics();
g2.setBackground(new Color(0,0,0,0));  // (The R, G, and B values
don't matter.)
g2.clearRect(0, 0, w, h);
```

(Note that simply drawing with a transparent color will not make pixels in the image transparent. The alpha component of a *Color* makes the color transparent when it is used for drawing; it does not change the transparency of the pixels that are modified by the drawing operation.)

As an example, just for fun, here is a method that will set the cursor of a component to be a red square with a transparent interior:

```
private void useRedSquareCursor() {
   BufferedImage image = new
BufferedImage(24,24,BufferedImage.TYPE_INT_ARGB);
   Graphics2D g2 = image.createGraphics();
   g2.setBackground(new Color(0,0,0,0));
   g2.clearRect(0, 0, 24, 24);  // (should not be necessary in a
new image)
   g2.setColor(Color.RED);
   g2.drawRect(0,0,23,23); // draw a red border of width 3 around
the square
   g2.drawRect(1,1,21,21);
   g2.drawRect(2,2,19,19);
   g2.dispose();
   Point hotSpot = new Point(12,12);
   Toolkit tk = Toolkit.getDefaultToolkit();
   Cursor cursor = tk.createCustomCursor(image,hotSpot,"square");
   setCursor(cursor);
}
```

---

### 13.2.3 Antialiasing

To draw a geometric figure such as a line or circle, you just have to color the pixels that are part of the figure, right? Actually, there is a problem with this. Pixels are little squares. Geometric figures, on the other hand, are made of geometric points that have no size at all. Think about drawing a circle, and think about a pixel on the boundary of that circle. The infinitely thin geometric boundary of the circle cuts through the pixel. Part of the pixel lies inside the circle, part lies outside. So, when we are filling the circle with color, do we color that pixel or not? A possible solution is to color the pixel if the geometric circle covers 50% or more of the pixel. Following this procedure, however, leads to a visual defect known as aliasing. It is visible in images as a jaggedness or "staircasing" effect along the borders of curved shapes. Lines that are not horizontal or vertical also have a jagged, aliased appearance. (The term "aliasing" seems to refer to the fact that many different geometric points map to the same pixel. If you think of the real-number coordinates of a geometric point as a "name" for the pixel that contains that point, then each pixel has many different names or "aliases.")

It's not possible to build a circle out of squares, but there is a technique that can eliminate some of the jaggedness of aliased images. The technique is called antialiasing. Antialiasing is based on transparency. The idea is simple: If 50% of a pixel is covered by the geometric figure that you are trying to draw, then color that pixel with a color that is 50% transparent. If 25% of the pixel is covered, use a color that is 75% transparent (25% opaque). If the entire pixel is covered by the figure, of course, use a color that is 100% opaque -- antialiasing only affects pixels that are only partly covered by the geometric shape.

In antialiasing, the color that you are drawing with is blended with the original color of the pixel, and the amount of blending depends on the fraction of the pixel that is covered by the geometric shape. (The fraction is difficult to compute exactly, so in practice, various methods are used to approximate it.) Of course, you still don't get a picture of the exact geometric shape, but antialiased images do tend to look better than jagged, aliased images.

For an example, look at the picture in the next subsection. Antialiasing is used to draw the panels in the second and third row of the picture, but it is not used in the top row. You should note the jagged appearance of the lines in the top row. (By the way, when antialiasing is applied to a line, the line is treated as a geometric rectangle whose width is equal to the line width.)

Antialiasing is supported in *Graphics2D*. By default, antialiasing is turned off. If `g2` is a graphics context of type *Graphics2D*, you can turn on antialiasing in `g2` by saying:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,

RenderingHints.VALUE_ANTIALIAS_ON);
```

As you can see, this is only a "hint" that you would like to use antialiasing, and it is even possible that the hint will be ignored. However, it is likely that subsequent drawing operations in g2 will be antialiased. If you want to turn antialiasing off in g2, you should say:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,

RenderingHints.VALUE_ANTIALIAS_OFF);
```

### 13.2.4 Strokes and Paints

When using the *Graphics* class, any line that you draw will be a solid line that is one pixel thick. The *Graphics2D* class makes it possible to draw a much greater variety of lines. You can draw lines of any thickness, and you can draw lines that are dotted or dashed instead of solid.

An object of type *Stroke* contains information about how lines should be drawn, including how thick the line should be and what pattern of dashes and dots, if any, should be used. Every *Graphics2D* has an associated *Stroke* object. The default *Stroke* draws a solid line of thickness one. To get lines with different properties, you just have to install a different stroke into the graphics context.

*Stroke* is an interface, not a class. The class *BasicStroke*, which implements the Stroke interface, is the one that is actually used to create stroke objects. For example, to create a stroke that draws solid lines with thickness equal to 3, use:

```
BasicStroke line3 = new BasicStroke(3);
```

If g2 is of type *Graphics2D*, the stroke can be installed in g2 by calling its setStroke() command:

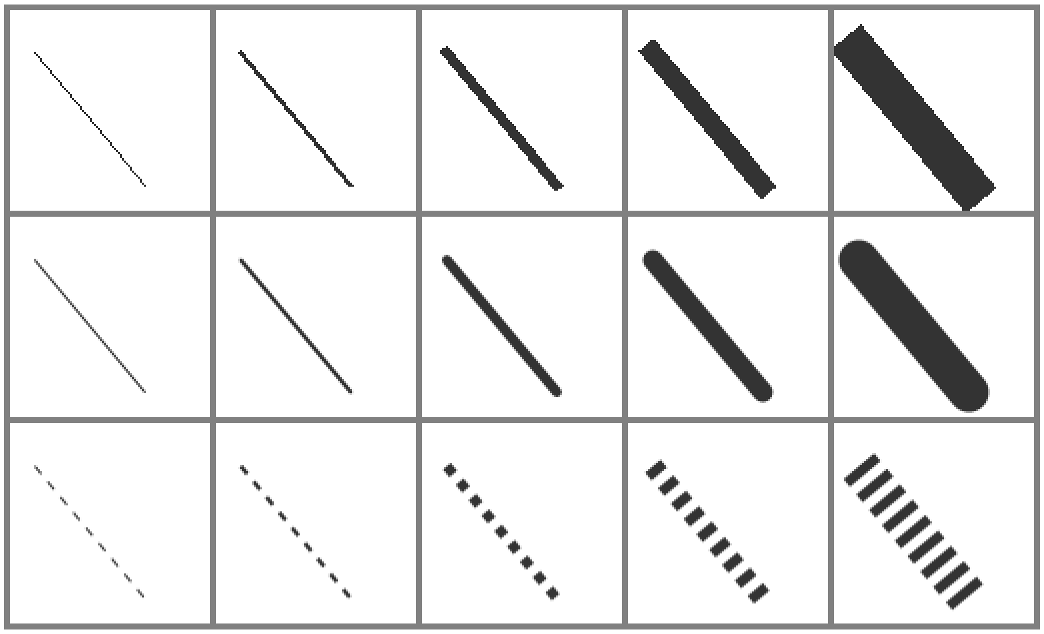
```
g2.setStroke(line3)
```

After calling this method, subsequent drawing operations will use lines that are three times as wide as the usual thickness. The thickness of a line can be given by a value of type float, not just by an int. For example, to use lines of thickness 2.5 in the graphics context g2, you can say:

```
g2.setStroke( new BasicStroke(2.5F) );
```

(Fractional widths make more sense if antialiasing is turned on.)

When you have a thick line, the question comes up, what to do at the ends of the line. If you draw a physical line with a large, round piece of chalk, the ends of the line will be rounded. When you draw a line on the computer screen, should the ends be rounded, or should the line simply be cut off flat? With the *BasicStroke* class, the choice is up to you. Maybe it's time to look at examples. This illustration shows fifteen lines, drawn using different *BasicStrokes*. Lines in the middle row have rounded ends; lines in the other two rows are simply cut off at their endpoints. Lines of various thicknesses are shown, and the bottom row shows dashed lines. (And, as mentioned above, only the bottom two rows are antialiased.)



This illustration is a screenshot from the sample program *StrokeDemo.java*. In that program, you can click and drag in any of the small panels, and the lines in all the panels will be redrawn as you move the mouse. In addition, if you right-click and drag, then rectangles will be drawn instead of lines; this shows that strokes are used for drawing the outlines of shapes and not just for straight lines. If you look at the corners of the rectangles that are drawn by the program, you'll see that there are several ways of drawing a corner where two wide line segments meet.

All the options that you want for a *BasicStroke* have to be specified in the constructor. Once the stroke object is created, there is no way to change the options. There is one constructor that lets you specify all possible options:

```
public BasicStroke( float width, int capType, int joinType, float
miterlimit,
                                        float[] dashPattern, float
dashPhase )
```

I don't want to cover all the options in detail, but here's some basic info:

- `width` specifies the thickness of the line
- `capType` specifies how the ends of a line are "capped." The possible values are `BasicStroke.CAP_SQUARE`, `BasicStroke.CAP_ROUND` and `BasicStroke.CAP_BUTT`. These values are used, respectively, in the first, second, and third rows of the above picture. The default is `BasicStroke.CAP_SQUARE`.
- `joinType` specifies how two line segments are joined together at corners. Possible values are `BasicStroke.JOIN_MITER`, `BasicStroke.JOIN_ROUND`, and `BasicStroke.JOIN_BEVEL`. Again, these are used in the three rows of panels in the sample

program; you will only see the effect if you run the program and draw some rectangles. The default is `BasicStroke.JOIN_MITER`.

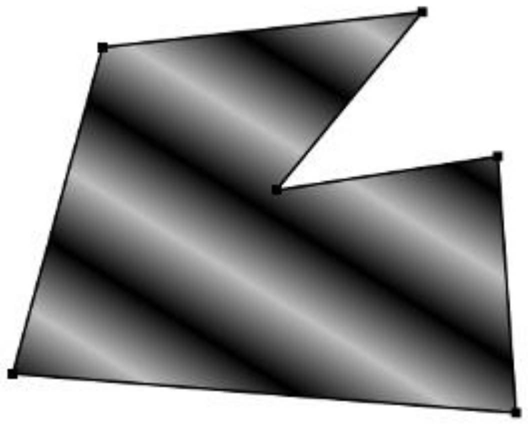
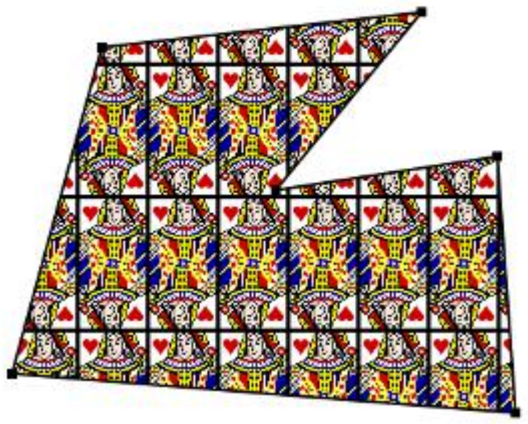
- `miterLimit` is used only if the value of `joinType` is `JOIN_MITER`; just use the default value, `10.0F`.
- `dashPattern` is used to specify dotted and dashed lines. The values in the array specify lengths in the dot/dash pattern. The numbers in the array represent the length of a solid piece, followed by the length of a transparent piece, followed by the length of a solid piece, and so on. At the end of the array, the pattern wraps back to the beginning of the array. If you want a solid line, use a different constructor that has fewer parameters.
- `dashPhase` tells the computer where to start in the `dashPattern` array, for the first segment of the line. Use 0 for this parameter in most cases.

For the third row in the above picture, the `dashPattern` is set to `new float[] {5,5}`. This means that the lines are drawn starting with a solid segment of length 5, followed by a transparent section of length 5, and then repeating the same pattern. A simple dotted line would have thickness 1 and `dashPattern` `new float[] {1,1}`. A pattern of short and long dashes could be made by using `new float[] {10,4,4,4}`. For more information, see the Java documentation, or try experimenting with the source code for the sample program.

---

So now we can draw fancier lines. But any drawing operation is still restricted to drawing with a single color. We can get around that restriction by using *Paint*. An object of type *Paint* is used to assign color to each pixel that is "hit" by a drawing operation. *Paint* is an `interface`, and the *Color* class implements the *Paint* interface. When a color is used for painting, it applies the same color to every pixel that is hit. However, there are other types of paint where the color that is applied to a pixel depends on the coordinates of that pixel. Standard Java includes several classes that define paint with this property: *TexturePaint* and several types of gradient paint. In a texture, the pixel colors come from an image, which is repeated, if necessary, like a wallpaper pattern to cover the entire xy-plane. In a gradient, the color that is applied to pixels changes gradually from one color to another color as you move from point to point. Java has three types of gradient paints: *GradientPaint*, *LinearGradientPaint*, and *RadialGradientPaint*.

It will be helpful to look at some examples. This illustration shows a polygon filled with two different paints. The polygon on the left uses a *GradientPaint* while the one on the right uses a *TexturePaint*. Note that in this picture, the paint is used only for filling the polygon. The outline of the polygon is drawn in a plain black color. However, *Paint* objects can be used for drawing lines as well as for filling shapes. These pictures were made by the sample program *PaintDemo.java*. In that program, you can select among several different paints, and you can control certain properties of the paints.

Basic gradient paints are created using the constructor

```
public GradientPaint(float x1, float y1, Color c1,
                                 float x2, float y2, Color c2, boolean
cyclic)
```

This constructs a gradient that has color `c1` at the point with coordinates `(x1,y1)` and color `c2` at the point `(x2,y2)`. As you move along the line between the two points, the color of the gradient changes from `c1` to `c2`; along lines perpendicular to this line, the color is constant. The last parameter, `cyclic`, tells what happens if you move past the point `(x2,y2)` on the line from `(x1,y1)` to `(x2,y2)`. If `cyclic` is `false`, the color stops changing and any point beyond `(x2,y2)` has color `c2`. If `cyclic` is `true`, then the colors continue to change in a cyclic pattern after you move past `(x2,y2)`. (It works the same way if you move past the other endpoint, `(x1,y1)`.) In most cases, you will set `cyclic` to `true`. Note that you can vary the points `(x1,y1)` and `(x2,y2)` to change the width and direction of the gradient. For example, to create a cyclic gradient that varies from black to light gray along the line from `(0,0)` to `(100,100)`, use:

```
new GradientPaint( 0, 0, Color.BLACK, 100, 100, Color.LIGHT_GRAY,
true)
```

The other two gradient paint classes, *LinearGradientPaint* and *RadialGradientPaint*, are more sophisticated. Linear gradient paints are similar to *GradientPaint* but can be based on more than two colors. Radial gradients color pixels based on their distance from a central point, which produces rings of constant color instead of lines of constant color. See the API documentation for details.

---

To construct a *TexturePaint*, you need a *BufferedImage* that contains the image that will be used for the texture. You also specify a rectangle in which the image will be drawn. The image will be scaled, if necessary, to exactly fill the rectangle. Outside the specified rectangle, the image will be repeated horizontally and vertically to fill the plane. You can vary the size and position of the rectangle to change the scale of the texture and its positioning on the plane. Ordinarily, however the upper left corner of the rectangle is placed at `(0,0)`, and the size of the rectangle is the same as the actual size of the image. The constructor for *TexturePaint* is defined as

```
public TexturePaint( BufferedImage textureImage, Rectangle2D
anchorRect)
```

The *Rectangle2D* class is part of the *Graphics2D* framework and will be discussed at the end of this section. Often, a call to the constructor takes the following form, which will show the image at its actual size:

```
new TexturePaint( image,
            new
Rectangle2D.Double(0,0,image.getWidth(),image.getHeight() )
```

Once you have a *Paint* object, you can use the `setPaint()` method of a *Graphics2D* object to install the paint in a graphics context. For example, if `g2` is of type *Graphics2D*, then the command

```
g2.setPaint( new
GradientPaint(0,0,Color.BLUE,100,100,Color.GREEN,true) );
```

sets up `g2` to use a gradient paint. Subsequent drawing operations with `g2` will draw using a blue/green gradient.

---

### 13.2.5 Transforms and Shapes

In the standard drawing coordinates on a component, the upper left corner of the component has coordinates `(0,0)`. Coordinates are integers, and the coordinates `(x,y)` refer to the point that is `x` pixels over from the left edge of the component and `y` pixels down from the top. With *Graphics2D*, however, you are not restricted to using these coordinates. In fact, you can can set up a *Graphics2D* graphics context to use any system of coordinates that you like. You can use this capability to select the coordinate system that is most appropriate for the things that you want to draw. For example, if you are drawing architectural blueprints, you might use coordinates in which one unit represents an actual distance of one foot.

Changes to a coordinate system are referred to as transforms. There are three basic types of transform. A translate transform changes the position of the origin, `(0,0)`. A scale transform changes the scale, that is, the unit of distance. And a rotation transform applies a rotation about some point. You can make more complex transforms by combining transforms of the three basic types. For example, you can apply a rotation, followed by a scale, followed by a translation, followed by another rotation. When you apply several transforms in a row, their effects are cumulative. It takes a fair amount of study to fully understand complex transforms, and transforms are a major topic in a course in computer graphics. I will limit myself here to discussing a few of the most simple cases, just to give you an idea of what transforms can do.

Suppose that g2 is of type *Graphics2D*. Then g2.translate(x,y) moves the origin, (0,0), to the point (x,y). This means that if you use coordinates (0,0) **after** saying g2.translate(x,y), then you are referring to the point that *used to be* (x,y), before the translation was applied. All other coordinate pairs are moved by the same amount. For example saying

```
g.translate(x,y);
g.drawLine( 0, 0, 100, 200 );
```

draws the same line as

```
g.drawLine( x, y, 100+x, 200+y );
```

In the second case, you are just doing the same translation "by hand." A translation (like all transforms) affects all subsequent drawing operations. Instead of thinking in terms of coordinate systems, you might find it clearer to think of what happens to the objects that are drawn. After you say g2.translate(x,y), any objects that you draw are displaced x units horizontally and y units vertically. Note that the parameters x and y can be real numbers.

As an example, perhaps you would prefer to have (0,0) at the center of a component, instead of at its upper left corner. To do this, just use the following command in the paintComponent() method of the component:

```
g2.translate( getWidth()/2, getHeight()/2 );
```

To apply a scale transform to a *Graphics2D* g2, use g2.scale(s,s), where s is the real number that specifies the scaling factor. If s is greater than 1, everything is magnified by a factor of s, while if s is between 0 and 1, everything is shrunk by a factor of s. The center of scaling is (0,0). That is, the point (0,0) is unaffected by the scaling, and other points more towards or away from (0,0) by a factor of s. Again, it can be clearer to think of the effect on objects that are drawn after a scale transform is applied. Those objects will be magnified or shrunk by a factor of s. Note that scaling affects **everything**, including thickness of lines and size of fonts. It is possible to use different scale factors in the horizontal and vertical direction with a command of the form g2.scale(sx,sy), although that will distort the shapes of objects. By the way, it is even possible to use scale factors that are less than 0, which results in reflections. For example, after calling g2.scale(-1,1), objects will be reflected horizontally through the line x=0.

The third type of basic transform is rotation. The command g2.rotate(r) rotates all subsequently drawn objects through an angle of r about the point (0,0). You can rotate instead about the point (x,y) with the command g2.rotate(r,x,y). All the parameters can be real numbers. Angles are measured in radians, where π radians are equal to 180 degrees. To rotate through an angle of d degrees, use

```
g2.rotate( d * Math.PI / 180 );
```

Positive angles are clockwise rotations, while negative angles are counterclockwise (unless you have applied a negative scale factor, which reverses the orientation).

Rotation is not as common as translation or scaling, but there are a few things that you can do with it that can't be done any other way. For example, you can use it to draw an image "on the slant." Rotation also makes it possible to draw text that is rotated so that its baseline is slanted or even vertical. To draw the string "Hello World" with its basepoint at `(x,y)` and rising at an angle of 30 degrees, use:

```
g2.rotate( -30 * Math.PI / 180, x, y );
g2.drawString( "Hello World", x, y );
```

To draw the message vertically, with the **center** of its baseline at the point `(x,y)`, we can use *FontMetrics* to measure the string, and say:

```
FontMetrics fm = g2.getFontMetrics( g2.getFont() );
int baselineLength = fm.stringWidth("Hello World");
g2.rotate( -90 * Math.PI / 180, x, y);
g2.drawString( "Hello World", x - baselineLength/2, y );
```

---

The drawing operations in the *Graphics* class use integer coordinates only. *Graphics2D* makes it possible to use real numbers as coordinates. This becomes particularly important once you start using transforms, since after you apply a scaling transform, a square of size one might cover many pixels instead of just a single pixel. Unfortunately, the designers of Java couldn't decide whether to use numbers of type float or double as coordinates, and their indecision makes things a little more complicated than they need to be. (My guess is that they really wanted to use float, since values of type float have enough accuracy for graphics and are probably used in the underlying graphical computations of the computer. However, in Java programming, it's easier to use double than float, so they wanted to make it possible to use double values too.)

To use real number coordinates, you have to use classes defined in the package `java.awt.geom`. Among the classes in this package are classes that represent geometric shapes such as lines and rectangles. For example, the class *Line2D* represents a line whose endpoints are given as real number coordinates. The unfortunate thing is that *Line2D* is an abstract class, which means that you can't create objects of type *Line2D* directly. However, *Line2D* has two concrete subclasses that can be used to create objects. One subclass uses coordinates of type float, and one uses coordinates of type double. The most peculiar part is that these subclasses are defined as static nested classes inside *Line2D*. Their names are *Line2D.Float* and *Line2D.Double*. This means that *Line2D* objects can be created, for example, with:

```
Line2D line1 = new Line2D.Float( 0.17F, 1.3F, -2.7F, 5.21F );
Line2D line2 = new Line2D.Double( 0, 0, 1, 0);
Line2D line3 = new Line2D.Double( x1, y1, x2, y2 );
```

where `x1`, `y1`, `x2`, `y2` are any numeric variables. In my own code, I generally use *Line2D.Double* rather than *Line2D.Float*.

Other shape classes in `java.awt.geom` are similar. The class that represents rectangles is *Rectangle2D*. To create a rectangle object, you have to use either *Rectangle2D.Float* or *Rectangle2D.Double*. For example,

```
Rectangle2D rect = new Rectangle2D.Double( -0.5, -0.5, 1.0, 1.0 );
```

creates a rectangle with a corner at `(-0.5,-0.5)` and with width and height both equal to 1. Other classes include *Point2D*, which represents a single point; *Ellipse2D*, which represents an oval; and *Arc2D*, which represents an arc of a circle.

If `g2` is of type *Graphics2D* and `shape` is an object belonging to one of the 2D shape classes, then the command

```
g2.draw(shape);
```

draws the shape. For a shape such as a rectangle or ellipse that has an interior, only the outline is drawn. To fill in the interior of such a shape, use

```
g2.fill(shape)
```

For example, to draw a line from `(x1,y1)` to `(x2,y2)`, use

```
g2.draw( new Line2D.Double(x1,y1,x2,y2) );
```

and to draw a filled rectangle with a corner at `(3.5,7)`, with width 5 and height 3, use

```
g2.fill( new Rectangle2D.Double(3.5, 7, 5, 3) );
```

The package `java.awt.geom` also has a very nice class *Path2D* that can be used to draw polygons and curves defined by any number of points. See the Java documentation if you want to find out how to use it.

This section has introduced you to many of the interesting features of *Graphics2D*, but there is still a large part of the *Graphics2D* framework for you to explore.

## Actions and Buttons

---

FOR THE PAST TWO SECTIONS, we have been looking at some of the more advanced aspects of the Java graphics API. But the heart of most graphical user interface programming is using GUI components. In this section and the next, we'll be looking at *JComponents*. We'll cover several component classes that were not covered in Chapter 6, as well as some additional features of classes that were covered there.

This section is mostly about buttons. Buttons are among the simplest of GUI components, and it seems like there shouldn't be all that much to say about them. However, buttons are not as simple as they seem. For one thing, there are many different types of buttons. The basic functionality of buttons in Java is defined by the class `javax.swing.AbstractButton`. Subclasses of this class represent push buttons, check boxes, and radio buttons. Menu items are also considered to be buttons. The *AbstractButton* class defines a surprisingly large API for controlling the appearance of buttons. This section will cover part of that API, but you should see the class documentation for full details.

In this section, we'll also encounter a few classes that do not themselves define buttons but that are related to the button API, starting with "actions."

---

### 13.3.1 Action and AbstractAction

The *JButton* class represents push buttons. Up until now, we have created push buttons using the constructor

```
public JButton(String text);
```

which specifies text that will appear on the button. We then added an *ActionListener* to the button, to respond when the user presses it. Another way to create a *JButton* is using an *Action*. The *Action* interface represents the general idea of some action that can be performed, together with properties associated with that action, such as a name for the action, an icon that represents the action, and whether the action is currently enabled or disabled. *Actions* are usually defined using the class *AbstractAction*, an `abstract` class which includes a method

```
public void actionPerformed(ActionEvent evt)
```

that must be defined in any concrete subclass. Often, this is done in an anonymous inner class. For example, suppose that `display` is an object that has a *clear()* method. Then an *Action* object that represents the action "clear the display" can be defined as:

```
Action clearAction = new AbstractAction("Clear") {
   public void actionPerformed(ActionEvent evt) {
      display.clear();
   }
};
```

The parameter, `"Clear"`, in the constructor of the *AbstractAction* is the name of the action. Other properties can be set by calling the method `putValue(key,value)`, which is part of the *Action* interface. For example,

```
clearAction.putValue(Action.SHORT_DESCRIPTION, "Clear the
Display");
```

sets the SHORT_DESCRIPTION property of the action to have the value "Clear the Display". The key parameter in the putValue() method is usually given as one of several constants defined in the *Action* interface. As another example, you can change the name of an action by using Action.NAME as the key in the putValue() method.

Once you have an *Action*, you can use it in the constructor of a button. For example, using the action clearAction defined above, we can create the *JButton*

```
          JButton clearButton = new JButton( clearAction );
```

The name of the action will be used as the text of the button, and some other properties of the button will be taken from properties of the action. For example, if the SHORT_DESCRIPTION property of the action has a value, then that value is used as the tooltip text for the button. (The tooltip text appears when the user hovers the mouse over the button.) Furthermore, when you change a property of the action, the corresponding property of the button will also be changed. For example, changing the Action's property associated with the key Action.NAME will also change the text on the button.

The *Action* interface defines a setEnabled() method that is used to enable and disable the action. The clearAction action can be enabled and disabled by calling clearAction.setEnabled(true) and clearAction.setEnabled(false). When you do this, any button that has been created from the action is also enabled or disabled at the same time.

Now of course, the question is, **why** should you want to use *Actions* at all? One advantage is that using actions can help you to organize your code better. You can create separate objects that represent each of the actions that can be performed in your program. This represents a nice division of responsibility. Of course, you could do the same thing with individual *ActionListener* objects, but then you couldn't associate descriptions and other properties with the actions.

More important is the fact that *Actions* can also be used in other places in the Java API. You can use an *Action* to create a *JMenuItem* in the same way as for a *JButton*:

```
          JMenuItem clearCommand = new JMenuItem( clearAction );
```

A *JMenuItem*, in fact, is a kind of button and shares many of the same properties that a *JButton* can have. You can use the **same** *Action* to create both a button and a menu item (or even several of each if you want). Whenever you enable or disable the action or change its name, the button and the menu item will **both** be changed to match. If you change the NAME property of the action, the text of both the menu item and the button will be set to the new name of the action. If you disable the action, both menu item and button will be disabled. You can think of the button and the menu items as being two presentations of the *Action*, and you don't have to keep track of the button or menu item after you create them. You can do everything that you need to do by manipulating the *Action* object.

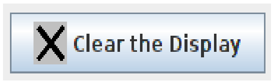
By the way, if you want to add a menu item that is defined by an *Action* to a menu, you don't even need to create the *JMenuItem* yourself. You can add the action object directly to the menu, and the menu item will be created from the properties of the action. For example, if `menu` is a *JMenu* and `clearAction` is an *Action*, you can simply say `menu.add(clearAction)`.

*Actions* have some other useful properties that we will encounter later in this section.

---

### 13.3.2 Icons on Buttons

In addition to -- or instead of -- text, buttons can also show icons. Icons are represented by the *Icon* interface and are usually created as *ImageIcons*, as discussed in Subsection 13.1.4. For example, here is a picture of a button that displays an image of a large "X" as its icon:



The icon for a button can be set by calling the button's `setIcon()` method, or by passing the icon object as a parameter to the constructor when the button is created. To create the button shown above, I created an *ImageIcon* from a *BufferedImage* on which I drew the picture that I wanted, and I constructed the *JButton* using a constructor that takes both the text and the icon for the button as parameters. Here's the code segment that does it:

```
BufferedImage image = new
BufferedImage(24,24,BufferedImage.TYPE_INT_RGB);

Graphics2D g2 = (Graphics2D)image.getGraphics();
g2.setColor(Color.LIGHT_GRAY);              // Draw the image for
the icon.
g2.fillRect(0,0,24,24);
g2.setStroke( new BasicStroke(3) );         //    Use thick lines.
g2.setColor(Color.BLACK);
g2.drawLine(4,4,20,20);                     //    Draw the "X".
g2.drawLine(4,20,20,4);
g2.dispose();

Icon clearIcon = new ImageIcon(image);      // Create the icon.

JButton clearButton = new JButton("Clear the Display", clearIcon);
```

You can create a button with an icon but no text by using a constructor that takes just the icon as parameter. Another alternative is for the button to get its icon from an *Action*. When a button is constructed from an action, it takes its icon from the value of the action property `Action.SMALL_ICON`. For example, suppose that we want to use an action named `clearAction` to create the button shown above. This could be done with:

```
clearAction.putValue( Action.SMALL_ICON, clearIcon );
JButton clearButton = new JButton( clearAction );
```

The icon could also be associated with the action by passing it as a parameter to the constructor of an *AbstractAction*:

```
Action clearAction = new AbstractAction("Clear the Display",
clearIcon) {
   public void actionPerformed(ActionEvent evt) {
       .
       .  // Carry out the action.
       .
   }
}
JButton clearButton = new JButton( clearAction );
```

The `SMALL_ICON` for an *Action* will also be used by a *JMenuItem* created brom the action. (An action can also have an icon associated with the key `Action.LARGE_ICON_KEY`. If it does, then a *JButton* will use the "large" icon in preference to the "small" icon. However, a *JMenuItem* will only use a small icon.)

The appearance of buttons can be tweaked in many ways. For example, you can change the size of the gap between the button's text and its icon. You can associate additional icons with a button that are used when the button is in certain states, such as when it is pressed or when it is disabled. It is even possible to change the positioning of the text with respect to the icon. For example, to place the text centered below the icon on a button, you can say:

```
button.setHorizontalTextPosition(JButton.CENTER);
button.setVerticalTextPosition(JButton.BOTTOM);
```

These methods and many others are defined in the class *AbstractButton*. This class is a superclass for *JMenuItem*, as well as for *JButton* and for the classes that define check boxes and radio buttons.

Finally, I will mention that it is possible to use icons on *JLabels* in much the same way that they can be used on *JButtons*. Placing an *ImageIcon* on a *JLabel* can be a convenient way to add a static image to your GUI.

---

### 13.3.3 Making Choices

The *JCheckBox* class was covered in Subsection 6.5.3, and the equivalent for use in menus, *JCheckBoxMenuItem*, in Subsection 6.7.1. A checkbox lets the user make a choice between two alternatives. A checkbox has two states, selected and not selected, and the user can change the state by clicking on the check box. The state of a checkbox can also be set programmatically by calling its `setSelected()` method, and the current value of the state can be checked using the `isSelected()` method.

Closely related to checkboxes are radio buttons. Like a checkbox, a radio button can be either selected or not. However, radio buttons are expected to occur in groups, and at most one radio button in a group can be selected at any given time. Radio button groups let the user make a choice among several alternatives. In Java, a radio button is represented by an object of type *JRadioButton*. When used in isolation, a *JRadioButton* acts just like a *JCheckBox*, and it has the same methods and events. Ordinarily, however, a *JRadioButton* is used in a group. A group of radio buttons is represented by an object belonging to the class *ButtonGroup*. A *ButtonGroup* is **not** a component and does not itself have a visible representation on the screen. A *ButtonGroup* works behind the scenes to organize a group of radio buttons, to ensure that at most one button in the group can be selected at any given time.

To use a group of radio buttons, you must create a *JRadioButton* object for each button in the group, and you must create one object of type *ButtonGroup* to organize the individual buttons into a group. Each *JRadioButton* must be added individually to some container, so that it will appear on the screen. (A *ButtonGroup* plays no role in the placement of the buttons on the screen.) Each *JRadioButton* must also be added to the *ButtonGroup*, which has an `add()` method for this purpose. If you want one of the buttons to be selected initially, you can call `setSelected(true)` for that button. If you don't do this, then none of the buttons will be selected until the user clicks on one of them.

As an example, here is how you could set up a set of radio buttons that can be used to select a color:

```
JRadioButton redRadio, blueRadio, greenRadio, yellowRadio;
         // Variables to represent the radio buttons.
         // These should probably be instance variables, so
         // that they can be used throughout the program.

ButtonGroup colorGroup = new ButtonGroup();

redRadio = new JRadioButton("Red");   // Create a button.
colorGroup.add(redRadio);             // Add it to the group.

blueRadio = new JRadioButton("Blue");
colorGroup.add(blueRadio);

greenRadio = new JRadioButton("Green");
colorGroup.add(greenRadio);

yellowRadio = new JRadioButton("Yellow");
colorGroup.add(yellowRadio);

redRadio.setSelected(true);  // Make an initial selection.
```

The individual buttons must still be added to a container if they are to appear on the screen. If you want to respond immediately when the user clicks on one of the radio buttons, you can register an *ActionListener* for each button. Just as for checkboxes, it is not always necessary to register listeners for radio buttons. In some cases, you can simply check the state of each button when you need to know it, using the button's `isSelected()` method.

You can add the equivalent of a group of radio buttons to a menu by using the class *JRadioButtonMenuItem*. To use this class, create several objects of this type, and create a

*ButtonGroup* to manage them. Add each *JRadioButtonMenuItem* to the *ButtonGroup*, and also add them to a *JMenu*. If you want one of the items to be selected initially, call its `setSelected()` method to set its selection state to true. You can add *ActionListeners* to each *JRadioButtonMenuItem* if you need to take some action when the user selects the menu item; if not, you can simply check the selected states of the buttons whenever you need to know them. As an example, suppose that `menu` is a *JMenu*. Then you can add a group of buttons to `menu` as follows:

```
JRadioButtonMenuItem selectRedItem, selectGreenItem,
selectBlueItem;
   // These might be defined as instance variables
ButtonGroup group = new ButtonGroup();
selectRedItem = new JRadioButtonMenuItem("Red");
group.add(selectRedItem);
menu.add(selectRedItem);
selectGreenItem = new JRadioButtonMenuItem("Green");
group.add(selectGreenItem);
menu.add(selectGreenItem);
selectBlueItem = new JRadioButtonMenuItem("Blue");
group.add(selectBlueItem);
menu.add(selectBlueItem);
```

---

When it's drawn on the screen, a *JCheckBox* includes a little box that is either checked or unchecked to show the state of the box. That box is actually a pair of *Icons*. One icon is shown when the check box is unselected; the other is shown when it is selected. You can change the appearance of the check box by substituting different icons for the standard ones.

The icon that is shown when the check box is unselected is just the main icon for the *JCheckBox*. You can provide a different unselected icon in the constructor or you can change the icon using the `setIcon()` method of the *JCheckBox* object. To change the icon that is shown when the check box is selected, use the `setSelectedIcon()` method of the *JCheckBox*. All this applies equally to *JRadioButton*, *JCheckBoxMenuItem*, and *JRadioButtonMenuItem*.

An example of this can be found in the sample program *ToolBarDemo.java*, which is discussed in the next subsection. That program creates a set of radio buttons that use custom icons. The buttons are created by the following method:

```
/**
 * Create a JRadioButton and add it to a specified button group.
The button
 * is meant for selecting a drawing color in the display.  The
color is used to
 * create two custom icons, one for the unselected state of the
button and one
 * for the selected state.  These icons are used instead of the
usual
 * radio button icons.
 * @param c the color of the button, and the color to be used for
drawing.
```

```
 *    (Note that c has to be "final" since it is used in the
anonymous inner
 *    class that defines the response to ActionEvents on the
button.)
 * @param grp the ButtonGroup to which the radio button will be
added.
 * @param selected if true, then the state of the button is set to
selected.
 * @return the radio button that was just created; sorry, but the
button
 *    is not as pretty as I would like!
 */
private JRadioButton makeColorRadioButton(final Color c,
                                          ButtonGroup grp, boolean
selected) {

   /* Create an ImageIcon for the normal, unselected state of the
button,
      using a BufferedImage that is drawn here from scratch. */

   BufferedImage image = new
BufferedImage(30,30,BufferedImage.TYPE_INT_RGB);
   Graphics g = image.getGraphics();
   g.setColor(Color.LIGHT_GRAY);
   g.fillRect(0,0,30,30);
   g.setColor(c);
   g.fill3DRect(1, 1, 24, 24, true);
   g.dispose();
   Icon unselectedIcon = new ImageIcon(image);

   /* Create an ImageIcon for the selected state of the button. */

   image = new BufferedImage(30,30,BufferedImage.TYPE_INT_RGB);
   g = image.getGraphics();
   g.setColor(Color.DARK_GRAY);
   g.fillRect(0,0,30,30);
   g.setColor(c);
   g.fill3DRect(3, 3, 24, 24, false);
   g.dispose();
   Icon selectedIcon = new ImageIcon(image);

   /* Create and configure the button. */

   JRadioButton button = new JRadioButton(unselectedIcon);
   button.setSelectedIcon(selectedIcon);
   button.addActionListener( new ActionListener() {
      public void actionPerformed(ActionEvent e) {
           // The action for this button sets the current drawing
color
           // in the display to c.
         display.setCurrentColor(c);
      }
   });
   grp.add(button);
   if (selected)
      button.setSelected(true);
   return button;
} // end makeColorRadioButton
```

---

It is possible to create radio buttons and check boxes -- both the regular sort and the corresponding menu items -- from *Actions*. The button or checkbox takes its name, main icon, tooltip text, and enabled/disabled state from the action. And the Action's `actionPerformed()` method is called when the user changes the state.Furthermore, an action has a property named `Action.SELECTED_KEY`, which means that the action itself can have a selected state. By default, the value of the `SELECTED_KEY` property is `null` and is not used for anything. However, you can set the value to be true or false by calling `action.putValue(Action.SELECTED_KEY,true)` or `action.putValue(Action.SELECTED_KEY,false)`. Once you have done so, the action's `SELECTED_KEY` property and the selected state of any checkbox or radio button created from the action will automatically be kept in sync. You can find out the current value of the property by calling `action.getValue(Action.SELECTED_KEY)`. Note that the value of a property is actually an object. When you set the value to `true` or `false`, the value is wrapped in an object of type *Boolean*. The return type of `action.getValue()` is *Object*, so you will probably need to type-cast the return value to *Boolean*.

---

Java has another component that lets the user choose from a set of options. the *JComboBox*. A combo box contains a list of items, but only displays the currently selected items. The user clicks the combo box to see a pop-up list of options and can then select from the list. The functionality is similar to a group of radio buttons.

*JComboBox* is a parameterized class, where the type parameter specifies the type of items that that combo box can hold. Most commonly, the items are strings, and the type is `JComboBox<String>`. This is the only case I will discuss. When a *JComboBox<String>* object is first constructed, it initially contains no items. An item is added to the bottom of the list of options by calling the combo box's instance method, `addItem(str)`, where `str` is the string that will be displayed in the menu.

For example, the following code will create an object of type *JComboBox* that contains the options Red, Blue, Green, and Black:

```
JComboBox<String> colorChoice = new JComboBox<String>();
colorChoice.addItem("Red");
colorChoice.addItem("Blue");
colorChoice.addItem("Green");
colorChoice.addItem("Black");
```

You can call the `getSelectedIndex()` method of a *JComboBox* to find out which item is currently selected. This method returns an integer that gives the position of the selected item in the list, where the items are numbered starting from zero. Alternatively, you can call `getSelectedItem()` to get the selected item itself. (This method returns a value of type *Object* -- even when the items are limited to being strings.) You can change the selection by calling the method `setSelectedIndex(n)`, where `n` is an integer giving the position of the item that you want to select.

The most common way to use a *JComboBox* is to call its `getSelectedIndex()` method when you have a need to know which item is currently selected. However, *JComboBox* components generate *ActionEvents* when the user selects an item, and you can register an *ActionListener* with the *JComboBox* if you want to respond to such events as they occur.

*JComboBoxes* have a nifty feature, which is probably not all that useful in practice. You can make a *JComboBox* "editable" by calling its method `setEditable(true)`. If you do this, the user can edit the selection by clicking on the *JComboBox* and typing. This allows the user to make a selection that is not in the pre-configured list that you provide. (The "Combo" in the name "JComboBox" refers to the fact that it's a kind of combination of menu and text-input box.) If the user has edited the selection in this way, then the `getSelectedIndex()` method will return the value `-1`, and `getSelectedItem()` will return the string that the user typed. An *ActionEvent* is triggered if the user presses return while typing in the *JComboBox*.

---

There is a lot of information in this section. The sample program *ChoiceDemo.java* demonstrates the use of combo boxes, check boxes, and radio buttons -- including the use of Actions with check boxes and radio buttons. I encourage you to run the program to see how these things work and to read the source code. The source code has a lot of comments to explain what is going on.

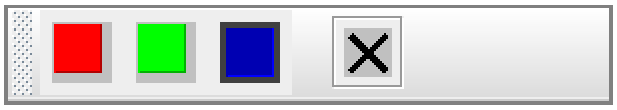
---

### 13.3.4 Toolbars

It has become increasingly common for programs to have a row of small buttons along the top or side of the program window that offer access to some of the commonly used features of the program. The row of buttons is known as a tool bar. Typically, the buttons in a tool bar are presented as small icons, with no text. Tool bars can also contain other components, such as *JTextFields* and *JLabels*.

In Swing, tool bars are represented by the class *JToolBar*. A *JToolBar* is a container that can hold other components. It is also itself a component, and so can be added to other containers. In general, the parent component of the tool bar should use a *BorderLayout*. The tool bar should occupy one of the edge positions -- `NORTH`, `SOUTH`, `EAST`, or `WEST` -- in the *BorderLayout*. Furthermore, the other three edge positions should be empty. The reason for this is that it might be possible (depending on the platform and configuration) for the user to drag the tool bar from one edge position in the parent container to another. It might even be possible for the user to drag the tool bar off its parent entirely, so that it becomes a separate window.

Here is a picture of a toolbar. This is from the sample program *ToolBarDemo.java*, which I discuss below:



In the demo program, the user can draw colored curves in a large drawing area. The first three buttons in the tool bar are a set of radio buttons that control the drawing color. The fourth button is a push button that the user can click to clear the drawing.

Tool bars are easy to use. You just have to create the *JToolBar* object, add it to a container, and add some buttons and possibly other components to the tool bar. One fine point is adding space to a tool bar, such as the gap between the radio buttons and the push button in the above picture. You can leave a gap by adding a separator to the tool bar. For example:

```
toolbar.addSeparator(new Dimension(20,20));
```

This adds an invisible 20-by-20 pixel block to the tool bar. This will appear as a 20 pixel gap between components.

Here is the constructor from the `ToolBarDemo` program. It shows how to create the tool bar and place it in a container. Note that class *ToolBarDemo* is a subclass of *JPanel*, and the tool bar and display are added to the panel object that is being constructed:

```
public ToolBarDemo() {

   setLayout(new BorderLayout(2,2));
   setBackground(Color.GRAY);
   setBorder(BorderFactory.createLineBorder(Color.GRAY,2));

   display = new Display();  // the area where the user draws.
   add(display, BorderLayout.CENTER);

   JToolBar toolbar = new JToolBar();
   add(toolbar, BorderLayout.NORTH);

   ButtonGroup group = new ButtonGroup();
   toolbar.add( makeColorRadioButton(Color.RED,group,true) );
   toolbar.add( makeColorRadioButton(Color.GREEN,group,false) );
   toolbar.add( makeColorRadioButton(Color.BLUE,group,false) );

   toolbar.addSeparator(new Dimension(20,20));

   toolbar.add( makeClearButton() );

}
```

If you want a vertical tool bar that can be placed in the `EAST` or `WEST` position of a *BorderLayout*, you should specify the orientation in the tool bar's constructor:

```
JToolBar toolbar = new JToolBar( JToolBar.VERTICAL );
```

The default orientation is `JToolBar.HORIZONTAL`. The orientation is adjusted automatically when the user drags the tool bar into a new position. If you want to prevent the user from dragging the tool bar, just say `toolbar.setFloatable(false)`.

---

### 13.3.5 Keyboard Accelerators

In most programs, commonly used menu commands have keyboard equivalents. The user can type the keyboard equivalent instead of selecting the command from the menu, and the result will be exactly the same. Typically, for example, the "Save" command has keyboard equivalent `CONTROL-S`, and the "Undo" command corresponds to `CONTROL-Z`. (Under Mac OS, the keyboard equivalents for these commands would probably be `META-S` and `META-Z`, where `META` refers to holding down the "apple" key.) The keyboard equivalents for menu commands are referred to as accelerators.

The class `javax.swing.KeyStroke` is used to represent key strokes that the user can type on the keyboard. Keystrokes can refer to pressing a key, releasing a key, or typing a character, possibly while holding down one or more of the modifier keys `control`, `shift`, `alt`, and `meta`. The *KeyStroke* class has a convenient static method, `getKeyStroke(String)`, that makes it easy to create key stroke objects representing key pressed events. For example,

```
KeyStroke.getKeyStroke( "ctrl S" )
```

returns a *KeyStroke* that represents the action of pressing the "S" key while holding down the control key. In addition to "ctrl", you can use the modifiers "shift", "alt", and "meta" in the string that describes the key stroke. You can even combine several modifiers, so that

```
KeyStroke.getKeyStroke( "ctrl shift Z" )
```

represents the action of pressing the "Z" key while holding down both the control and the shift keys. When the key stroke involves pressing a character key, the character must appear in the string in upper case form. You can also have key strokes that correspond to non-character keys. The number keys can be referred to as "1", "2", etc., while certain special keys have names such as "F1", "ENTER", and "LEFT" (for the left arrow key). The class *KeyEvent* defines many constants such as `VK_ENTER`, `VK_LEFT`, and `VK_S`. The names that are used for keys in the keystroke description are just these constants with the leading "`VK_`" removed.

There are at least two ways to associate a keyboard accelerator with a menu item. One is to use the `setAccelerator()` method of the menu item object:

```
JMenuItem saveCommand = new JMenuItem( "Save..." );
saveCommand.setAccelerator( KeyStroke.getKeyStroke("ctrl S") );
```

The other technique can be used if the menu item is created from an *Action*. The action property `Action.ACCELERATOR_KEY` can be used to associate a *KeyStroke* with an *Action*. When a menu item is created from the action, the keyboard accelerator for the menu item is taken from the value of this property. For example, if `redoAction` is an *Action* representing a "Redo" action, then you might say:

```
redoAction.putValue( Action.ACCELERATOR_KEY,
                              KeyStroke.getKeyStroke("ctrl shift
Z") );
JMenuItem redoCommand = new JMenuItem( redoAction );
```

or, alternatively, you could simply add the action to a *JMenu*, `editMenu`, with `editMenu.add(redoAction)`. (Note, by the way, that accelerators apply only to menu items, not to push buttons. When you create a *JButton* from an action, the `ACCELERATOR_KEY` property of the action is ignored.)

Note that you can use accelerators for *JCheckBoxMenuItems* and *JRadioButtonMenuItems*, as well as for simple *JMenuItems*.

For examples of using keyboard accelerators, see *ChoiceDemo.java* and the solution to Exercise 13.2.

---

By the way, as noted above, in the Mac OS operating system, the meta (or apple) key is usually used for keyboard accelerators instead of the control key. Java has a way of determining the correct modifier for the computer on which it is running. However, it requires the use of a different method for constructing the *KeyStroke*. The function call

```
int shortcutMask =
Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()
```

returns a "mask" that can be used to represent the correct modifier key when the keystroke is created by the function

```
KeyStroke stoke = KeyStroke( keyCode, shortcutMask );
```

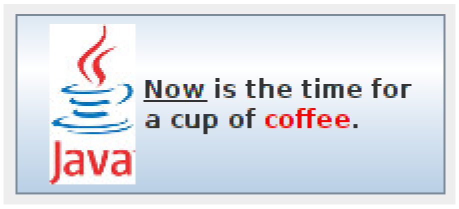
The first parameter, `keyCode`, is one of the constants such as `KeyEvent.VK_Z` that represent keys on the keyboard. For example, `KeyStroke.getKeyStroke(KeyEvent.VK_Z,shortcutMask)` represents the action of pressing the Z key while holding down the meta key on a Mac or the control key on Windows or Linux. It is possible to add other modifiers to the `shortcutMask` to account for holding down the shift or alt keys; see the documentation.

As an example, here is a code segment that is used in *ChoiceDemo.java* to make a "Quit" menu item that will have the appropriate modifier key for the computer on which the program is running

```
int shortcutMask =
Toolkit.getDefaultToolkit().getMenuShortcutKeyMask();
KeyStroke quitKeyStroke = KeyStroke.getKeyStroke(KeyEvent.VK_Q,
shortcutMask);
JMenuItem quit = new JMenuItem("Quit");
quit.setAccelerator(quitKeyStroke);
quit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        System.exit(0);
    }
});
controlMenu.add(quit);
```

### 13.3.6 HTML on Buttons

As a final stop in this brief tour of ways to spiff up your buttons, I'll mention the fact that the text that is displayed on a button can be specified in HTML format. HTML is the markup language that is used to write web pages. I don't cover it in this book., but I will explain a few things that you can do with it. HTML allows you to apply color or italics or other styles to just part of the text on your buttons. It also makes it possible to have buttons that display multiple lines of text. (You can also use HTML on *JLabels*, which can be even more useful.) Here's a picture of a button with HTML text (along with a "Java" icon):



If the string of text that is applied to a button starts with "`<html>`", then the string is interpreted as HTML. The string does not have to use strict HTML format; for example, you don't need a closing `</html>` at the end of the string. To get multi-line text, use `<br>` in the string to represent line breaks. If you would like the lines of text to be center justified, include the entire text (except for the `<html>`) between `<center>` and `</center>`. For example,

```
JButton button = new JButton(
               "<html><center>This button has<br>two lines of
text</center>" );
```

creates a button that displays two centered lines of text. You can apply italics to part of the string by enclosing that part between `<i>` and `</i>`. Similarly, use `<b>...</b>` for bold text and `<u>...</u>` for underlined text. For green text, enclose the text between `<font color=green>` and `</font >`. You can, of course, use other colors in place of "green." The "Java" button that is shown above was created using:

```
JButton javaButton = new JButton( "<html><u>Now</u> is the time
for<br>" +
                                "a nice cup of <font
          color=red>coffee</font>." );
```

and then adding the Java image as an icon for the button.

# Complex Components and MVC

---

SINCE EVEN BUTTONS turn out to be pretty complex, as seen in the previous section, you might guess that there is a lot more complexity lurking in the Swing API. While this is true, a lot of that complexity works to your benefit as a programmer, since a lot of it is hidden in normal uses of Swing components. For example, you don't have to know about all the complex details of buttons in order to use them effectively in most programs.

Swing defines several component classes that are much more complex than those we have looked at so far, but even the most complex components are not very difficult to use for many purposes. In this section, we'll look at components that support display and manipulation of lists, tables, and text documents. To use these complex components effectively, you'll need to know something about the Model-View-Controller pattern that is used as a basis for the design of many Swing components. This pattern is discussed in the first part of this section.

This section is our last look at Swing components, but there are a number of component classes that have not even been touched on in this book. Some useful ones that you might want to look into include: *JTabbedPane*, *JSplitPane*, *JTree*, *JSpinner*, *JPopupMenu*, *JProgressBar*, and *JScrollBar*.

At the end of the section, we'll look briefly at the idea of writing custom component classes -- something that you might consider when even the large variety of components that are already defined in Swing don't do quite what you want (or when they do too much, and you want something simpler).

---

### 13.4.1 Model-View-Controller

One of the principles of object-oriented design is division of responsibilities. Ideally, an object should have a single, clearly defined role, with a limited realm of responsibility. One application of this principle to the design of graphical user interfaces is the MVC pattern. "MVC" stands for "Model-View-Controller" and refers to three different realms of responsibility in the design of a graphical user interface.

When the MVC pattern is applied to a component, the model consists of the data that represents the current state of the component. The view is simply the visual presentation of the component on the screen. And the controller is the aspect of the component that carries out actions in response to events generated by the user (or by other sources such as timers). The idea is to assign responsibility for the model, the view, and the controller to different objects.

The view is the easiest part of the MVC pattern to understand. It is often represented by the component object itself, and its responsibility is to draw the component on the screen. In doing this, of course, it has to consult the model, since the model represents the state of the component, and that state can determine what appears on the screen. To get at the model data -- which is stored in a separate object according to the MVC pattern -- the component object needs to keep a reference to the model object. Furthermore, when the model changes, the view often needs to be redrawn to reflect the changed state. The component needs some way of knowing when changes in the model occur. Typically, in Java, this is done with events and listeners. The model object is set up to generate events when its data changes. The view object registers itself as a listener for those events. When the model changes, an event is generated, the view is notified of that event, and the view responds by updating its appearance on the screen.

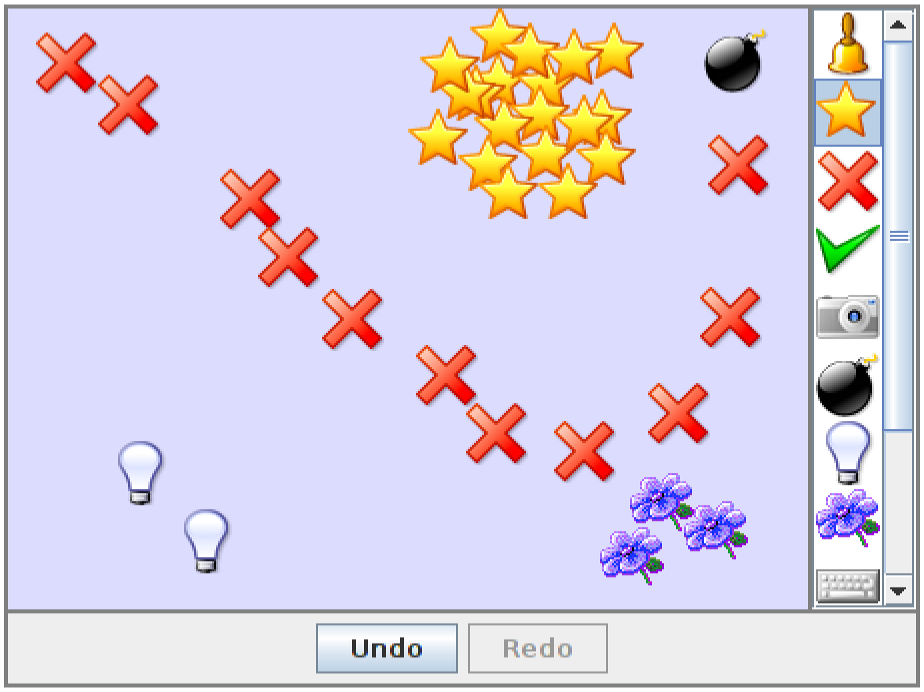
When MVC is used for Swing components, the controller is generally not so well defined as the model and view, and its responsibilities are often split among several objects. The controller might include mouse and keyboard listeners that respond to user events on the view; *Actions* that respond to menu commands or buttons; and listeners for other high-level events, such as those from a slider, that affect the state of the component. Usually, the controller responds to events by making modifications to the model, and the view is changed only indirectly, in response to the changes in the model.

The MVC pattern is used in many places in the design of Swing. It is even used for buttons. The state of a Swing button is stored in an object of type *ButtonModel*. The model stores such information as whether the button is enabled, whether it is selected, and what *ButtonGroup* it is part of, if any. If `button` is of type *JButton* (or one of the other subclasses of *AbstractButton*), then its *ButtonModel* can be obtained by calling `button.getModel()`. In the case of buttons, you might never need to use the model or even know that it exists. But for the list and table components that we will look at next, knowledge of the model is essential.

---

### 13.4.2 Lists and ListModels

A *JList* is a component that represents a list of items that can be selected by the user. The sample program *SillyStamper.java* allows the user to select one icon from a *JList* of *Icons*. The user selects an icon from the list by clicking on it. The selected icon can be "stamped" onto a drawing area by clicking on the drawing area. (The icons in this program are from the KDE desktop project.) Here is a picture of the program with several icons already stamped onto the drawing area and with the "star" icon selected:



Note that the scrollbar in this program is not part of the *JList*. To add a scrollbar to a list, the list must be placed into a *JScrollPane*. See Subsection 6.5.4, where the use of *JScrollPane* to hold a *JTextArea* was discussed. Scroll panes are used in the same way with lists and with other components. In this case, the *JList*, `iconList`, was added to a scroll pane and the scroll pane was added to a panel with the single command:

```
add( new JScrollPane(iconList), BorderLayout.EAST );
```

*JList* is a parameterized type, where the type parameter indicates what type of object can be displayed in the list. The most common types are `JList<String>` and `JList<Icon>`. Other types can be used, but they will be converted to strings for display. (It's possible to "teach" a *JList* how to display other types of items; see the `setCellRenderer()` method in the *JList* class.)

One way to construct a *JList* is from an array that contains the objects that will appear in the list. In the `SillyStamper` program, the images for the icons are read from resource files, the icons are placed into an array, and the array is used to construct a `JList<Icon>`. This is done by the following method:

```
private JList<Icon> createIconList() {

   String[] iconNames = new String[] {
      "icon5.png", "icon7.png", "icon8.png", "icon9.png",
"icon10.png",
      "icon11.png", "icon24.png", "icon25.png", "icon26.png",
"icon31.png",
      "icon33.png", "icon34.png"
   };               // Array containing resource file names for the
icon images.

   iconImages = new Image[iconNames.length];

   ClassLoader classLoader = getClass().getClassLoader();
   Toolkit toolkit = Toolkit.getDefaultToolkit();
   try {                        // Get the icon images from the
resource files.
      for (int i = 0; i < iconNames.length; i++) {
         URL imageURL = classLoader.getResource("stamper_icons/" +
iconNames[i]);
         if (imageURL == null)
            throw new Exception();
         iconImages[i] = toolkit.createImage(imageURL);
      }
   }
   catch (Exception e) {
      iconImages = null;
      return null;
   }

   ImageIcon[] icons = new ImageIcon[iconImages.length];
   for (int i = 0; i < iconImages.length; i++)  // Create the
icons.
      icons[i] = new ImageIcon(iconImages[i]);

   JList<Icon> list = new JList<Icon>(icons);  // A list containing
image icons.
   list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
   list.setSelectedIndex(0);   // First item in the list is
currently selected.

   return list;
}
```

By default, the user can select any number of items in a list. A single item is selected by clicking on it. Multiple items can be selected by shift-clicking and by either control-clicking or meta-clicking (depending on the platform). In the `SillyStamper` program, I wanted to restrict the selection so that only one item can be selected at a time. This restriction is imposed by calling

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

With this selection mode, when the user selects an item, the previously selected item, if any, is deselected. Note that the selection can be changed by the program by calling `list.setSelectedIndex(itemNum)`. Items are numbered starting from zero. To find out the currently selected item in single selection mode, call `list.getSelectedIndex()`. This returns the item number of the selected item, or `-1` if no item is currently selected. If multiple selections are allowed, you can call `list.getSelectedIndices()`, which returns an array of ints that contains the item numbers of all selected items.

Now, the list that you see on the screen is only the *view* aspect of the list. The *controller* consists of the listener objects that respond when the user clicks an item in the list. For its *model*, a *JList* uses an object of type *ListModel*. This is the object that knows the actual list of items. A model is defined not only by the data that it contains but by the set of operations that can be performed on the data. When a *JList* is constructed from an array of objects, the model that is used is very simple. The model can tell you how many items it contains and what those items are, but it can't do much else. In particular, there is no way to add items to the list or to delete items from the list! If you need that capability, you will have to use a different list model.

The class *DefaultListModel* defines list models that support adding items to and removing items from the list. Like *JList*, it is a parameterized class. (Note that the list model that you get when you create a *JList* from an array is **not** of this type.) If `dlmodel` is of type *DefaultListModel*, the following methods, among others, are defined:

- `dlmodel.getSize()` -- returns the number of items.
- `dlmodel.getElementAt(index)` -- returns the item at position `index` in the list.
- `dlmodel.addElement(item)` -- Adds `item` to the end of the list; `item` can be any *Object*.
- `dlmodel.insertElementAt(item, index)` -- inserts the specified `item` into the list at the specified `index`; items that come after that position in the list are moved down to make room for the new item.
- `dlmodel.setElementAt(item, index)` -- Replaces the item that is currently at position `index` in the list with `item`.
- `dlmodel.remove(index)` -- removes the item at position `index` in the list.
- `dlmodel.removeAllElements()` -- removes everything from the list, leaving it empty.

To use a modifiable `JList<T>`, you should create a `DefaultListModel<T>`, add any items to it that should be in the list initially, and pass it to the *JList* constructor. For example:

```
DefaultListModel<String> listModel;
JList<String> flavorList;

listModel = new DefaultListModel<String>();    // Create the model
object.

listModel.addElement("Chocolate");      // Add items to the model.
listModel.addElement("Vanilla");
listModel.addElement("Strawberry");
listModel.addElement("Rum Raisin");

flavorList = new JList<String>(listModel);     // Create the list
component.
```

By keeping a reference to the model around in an instance variable, you will be able to add and delete flavors as the program is running by calling the appropriate methods in `listModel`. Keep in mind that changes that are made to the *model* will automatically be reflected in the *view*. Behind the scenes, when a list model is modified, it generates an event of type *ListDataEvent*. The *JList* registers itself with its model as a listener for these events, and it responds to an event by redrawing itself to reflect the changes in the model. The programmer doesn't have to take any extra action, beyond changing the model.

By the way, the model for a *JList* actually has another part in addition to the *ListModel*: An object of type *ListSelectionModel* stores information about which items in the list are currently selected. When the model is complex, it's not uncommon to use several model objects to store different aspects of the state.

---

### 13.4.3 Tables and TableModels

Like a *JList*, a *JTable* displays a collection of items to the user. However, tables are much more complicated than lists. Perhaps the most important difference is that it is possible for the user to edit items in the table. Table items are arranged in a grid of rows and columns. Each grid position is called a cell of the table. Each column can have a header, which appears at the top of the column and contains a name for the column.

It is easy to create a *JTable* from an array that contains the names of the columns and a two-dimensional array that contains the items that go into the cells of the table. As an example, the sample program *StatesAndCapitalsTableDemo.java* creates a table with two columns named "State" and "Capital City." The first column contains a list of the states of the United States and the second column contains the name of the capital city of each state. The table can be created as follows:

```
String[][] statesAndCapitals = new String[][] {
           { "Alabama", "Montgomery" },
           { "Alaska", "Juneau" },
           { "Arizona", "Phoenix" },
                .
                .
                .
           { "Wisconsin", "Madison" },
           { "Wyoming", "Cheyenne" }
        };

String[] columnHeads = new String[] { "State", "Capital City" };

JTable table = new JTable(statesAndCapitals, columnHeads);
```

Since a table does not come with its own scroll bars, it is almost always placed in a *JScrollPane* to make it possible to scroll the table. For example:

```
add( new JScrollPane(table), BorderLayout.CENTER );
```

The column headers of a *JTable* are not actually part of the table; they are in a separate component. But when you add the table to a *JScrollPane*, the column headers are automatically placed at the top of the pane.

Using the default settings, the user can edit any cell in the table by clicking that cell. When editing a cell, the arrow keys can be used to move from one cell to another. The user can change the order of the columns by dragging a column header to a new position. The user can also change the width of the columns by dragging the line that separates neighboring column headers. You can try all this in *StatesAndCapitalsTableDemo.java*.

Allowing the user to edit all entries in the table is not always appropriate; certainly it's not appropriate in the "states and capitals" example. A *JTable* uses an object of type *TableModel* to store information about the contents of the table. The model object is also responsible for deciding whether or not the user should be able to edit any given cell in the table. *TableModel* includes the method

```
public boolean isCellEditable(int rowNum, columnNum)
```

where `rowNum` and `columnNum` are the position of a cell in the grid of rows and columns that make up the table. When the controller wants to know whether a certain cell is editable, it calls this method in the table model. If the return value is true, the user is allowed to edit the cell.

The default model that is used when the table is created from an array of objects allows editing of all cells. For this model, the return value of `isCellEditable()` is true in all cases. To make some cells non-editable, you have to provide a different model for the table. One way to do this is to create a subclass of *DefaultTableModel* and override the `isCellEditable()` method. (*DefaultTableModel* and some other classes that are discussed in this section are defined in the package `javax.swing.table`.) Here is how this might have been done in the "states and capitals" program to make all cells non-editable:

```
TableModel model = new
DefaultTableModel(statesAndCapitals,columnHeads) {
   public boolean isCellEditable(int row, int col) {
      return false;
   }
};
JTable table = new JTable(model);
```

Here, an anonymous subclass of *DefaultTableModel* is created in which the `isCellEditable()` method returns `false` in all cases, and the model object that is created from that class is passed as a parameter to the *JTable* constructor.

The *DefaultTableModel* class defines many methods that can be used to modify the table, including for example: `setValueAt(item,rowNum,colNum)` to change the item in a given cell; `removeRow(rowNum)` to delete a row; and `addRow(itemArray)` to add a new row at the end of the table that contains items from the array `itemArray`. Note that if the item in a given cell is set to `null`, then that cell will be empty. Remember, again, that when you modify the model, the view is automatically updated to reflect the changes.

In addition to the `isCellEditable()` method, the table model method that you are most likely to want to override is `getColumnClass()`, which is defined as

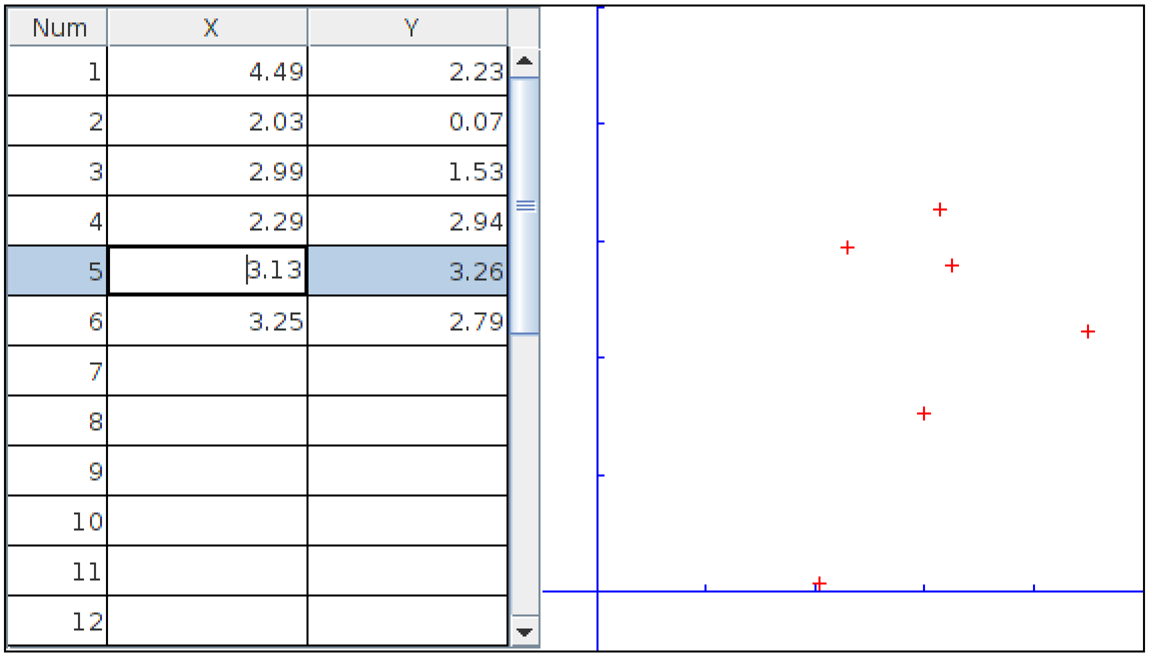
```
public Class<?> getColumnClass(columnNum)
```

The purpose of this method is to specify what kind of values are allowed in the specified column. The return value from this method is of type *Class*. (The "`<?>`" is there for technical reasons having to do with generic programming. See Section 10.5, but don't worry about understanding it here.) Although class objects have crept into this book in a few places -- in the discussion of *ClassLoaders* in Subsection 13.1.3 for example -- this is the first time we have directly encountered the class named *Class*. An object of type *Class* represents a class. A *Class* object is usually obtained from the name of the class using expressions of the form "`Double.class`" or "`JTable.class`". If you want a three-column table in which the column types are *String*, *Double*, and *Boolean*, you can use a table model in which `getColumnClass` is defined as:

```
public Class<?> getColumnClass(columnNum) {
   if (columnNum == 0)
      return String.class;
   else if (columnNum = 1)
      return Double.class;
   else
      return Boolean.class;
}
```

The table will call this method and use the return value to decide how to display and edit items in the table. For example, if a column is specified to hold *Boolean* values, the cells in that column will be displayed and edited as check boxes. For numeric types, the table will not accept illegal input when the user types in the value. (It is possible to change the way that a table edits or displays items. See the methods `setDefaultEditor()` and `setDefaultRenderer()` in the *JTable* class.)

As an alternative to using a subclass of *DefaultTableModel*, a custom table model can also be defined using a subclass of *AbstractTableModel*. Whereas *DefaultTableModel* provides a lot of predefined functionality, *AbstractTableModel* provides very little. However, using *AbstractTableModel* gives you the freedom to represent the table data any way you want. The sample program *ScatterPlotTableDemo.java* uses a subclass of *AbstractTableModel* to define the model for a *JTable*. In this program, the table has three columns. The first column holds a row number and is not editable. The other columns hold values of type *Double*; these two columns represent the x- and y-coordinates of points in the plane. The points themselves are graphed in a "scatter plot" next to the table. Initially, the program fills in the first six points with random values. Here is a picture of the program, with the x-coordinate in row 5 selected for editing:



| Num | X | Y |
|---|---|---|
| 1 | 4.49 | 2.23 |
| 2 | 2.03 | 0.07 |
| 3 | 2.99 | 1.53 |
| 4 | 2.29 | 2.94 |
| 5 | 3.13 | 3.26 |
| 6 | 3.25 | 2.79 |
| 7 | | |
| 8 | | |
| 9 | | |
| 10 | | |
| 11 | | |
| 12 | | |

Note, by the way, that in this program, the scatter plot can be considered to be a view of the table model, in the same way that the table itself is. The scatter plot registers itself as a listener with the model, so that it will receive notification whenever the model changes. When that happens, the scatter plot redraws itself to reflect the new state of the model. It is an important property of the MVC pattern that several views can share the same model, offering alternative presentations of the same data. The views don't have to know about each other or communicate with each other except by sharing the model. Although I didn't do it in this program, it would even be possible to add a controller to the scatter plot view. This could let the user add a point by clicking the mouse on the scatter plot or drag a point to change its coordinates. The controller would update the table model, based on the user actions. Since the scatter plot and table share the same model, the values displayed in the table would automatically change to match.

Here is the definition of the class that defines the model in the scatter plot program. All the methods in this class must be defined in any subclass of *AbstractTableModel* except for `setValueAt()`, which only has to be defined if the table is modifiable.

```
/**
 * This class defines the TableModel that is used for the JTable in
this
 * program.  The table has three columns.  Column 0 simply holds
the
 * row number of each row.  Column 1 holds the x-coordinates of the
 * points for the scatter plot, and Column 2 holds the y-
coordinates.
 * The table has 25 rows.  No support is provided for adding more
rows.
 */
private class CoordInputTableModel extends AbstractTableModel {

   private Double[] xCoord = new Double[25];  // Data for Column 1.
   private Double[] yCoord = new Double[25];  // Data for Column 2.
        // Initially, all the values in the array are null, which 
means
        // that all the cells are empty.

   public int getColumnCount() {  // Tells caller how many columns 
there are.
      return 3;
   }

   public int getRowCount() {  // Tells caller how many rows there 
are.
      return xCoord.length;
   }

   public Object getValueAt(int row, int col) {  // Get value from 
cell.
      if (col == 0)
         return (row+1);        // Column 0 holds the row number.
      else if (col == 1)
         return xCoord[row];    // Column 1 holds the x-
coordinates.
      else
         return yCoord[row];    // Column 2 holds the y-
coordinates.
   }

   public Class<?> getColumnClass(int col) {  // Get data type of 
column.
      if (col == 0)
         return Integer.class;
      else
         return Double.class;
   }

   public String getColumnName(int col) {  // Returns a name for 
column header.
      if (col == 0)
         return "Num";
      else if (col == 1)
         return "X";
      else
         return "Y";
   }

   public boolean isCellEditable(int row, int col) { // Can user 
edit cell?
      return col > 0;
   }

   public void setValueAt(Object obj, int row, int col) {
         // (This method is called by the system if the value of 
the cell
         // needs to be changed because the user has edited the 
cell.
         // It can also be called to change the value 
programmatically.
```

```
         // In this case, only columns 1 and 2 can be modified, and
the data
         // type for obj must be Double.  The method
fireTableCellUpdated()
         // has to be called to send an event to registered
listeners to
         // notify them of the modification to the table model.)
      if (col == 1)
         xCoord[row] = (Double)obj;
      else if (col == 2)
         yCoord[row] = (Double)obj;
      fireTableCellUpdated(row, col);
   }

}  // end nested class CoordInputTableModel
```

In addition to defining a custom table model, I customized the appearance of the table in several ways. Because this involves changes to the view, most of the changes are made by calling methods in the *JTable* object. For example, since the default height of the cells was too small for my taste, I called `table.setRowHeight(25)` to increase the height. To make lines appear between the rows and columns, I found that I had to call both `table.setShowGrid(true)` and `table.setGridColor(Color.BLACK)`. Some of the customization has to be done to other objects. For example, to prevent the user from changing the order of the columns by dragging the column headers, I had to use

```
table.getTableHeader().setReorderingAllowed(false);
```

Tables are quite complex, and I have only discussed a part of the table API here. Nevertheless, I hope that you have learned enough to start using them and to learn more about them on your own.

### 13.4.4 Documents and Editors

As a final example of complex components, we look briefly at *JTextComponent* and its subclasses. A *JTextComponent* displays text that can, optionally, be edited by the user. Two subclasses, *JTextField* and *JTextArea*, were introduced in Subsection 6.5.4. But the real complexity comes in another subclass, *JEditorPane*, that supports display and editing of styled text. This allows features such as boldface and italic. A *JEditorPane* can even work with basic HTML documents.

It is almost absurdly easy to write a simple web browser program using a *JEditorPane* (although the program that you get can only handle basic web pages and does a pretty bad job on many modern web pages). This is done in the sample program *SimpleWebBrowser.java*. In this program, the user enters the URL of a web page, and the program tries to load and display the web page at that location. A *JEditorPane* can handle pages with content type "text/plain", "text/html", and "text/rtf". (The content type "text/rtf" represents styled or "rich text format" text. URLs and content types were covered in Subsection 11.4.1.) If `editPane` is of type

*JEditorPane* and `url` is of type *URL*, then the statement "`editPane.setPage(url);`" is sufficient to load the page and display it. Since this can generate an exception, the following method is used in *SimpleWebBrowser.java* to display a page:

```
private void loadURL(URL url) {
   try {
      editPane.setPage(url);
   }
   catch (Exception e) {
      editPane.setContentType("text/plain"); // Set pane to display
plain text.
      editPane.setText( "Sorry, the requested document was not
found\n"
            +"or cannot be displayed.\n\nError:" + e);
   }
}
```

An HTML document can include links to other pages. When the user clicks on a link, the web browser should go to the linked page. A *JEditorPane* does not do this automatically, but it does generate an event of type *HyperLinkEvent* when the user clicks a link (provided that the edit pane has been set to be non-editable by the user). A program can register a listener for such events and respond by loading the new page.

There are a lot of web pages that a *JEditorPane* won't be able to display correctly, but it can be very useful in cases where you have control over the pages that will be displayed. A nice application is to distribute HTML-format help and information files with a program. The files can be stored as resource files in the jar file of the program, and a URL for a resource file can be obtained in the usual way, using the `getResource()` method of a *ClassLoader*. (See Subsection 13.1.3.)

It turns out, by the way, that *SimpleWebBrowser.java* is a little too simple. A modified version, *SimpleWebBrowserWithThread.java*, improves on the original by using a thread to load a page and by checking the content type of a page before trying to load it. It actually does work as a simple web browser. Try it!

The model for a *JTextComponent* is an object of type *Document*. If you want to be notified of changes in the model, you can add a listener to the model using

```
textComponent.getDocument().addDocumentListener(listener)
```

where `textComponent` is of type *JTextComponent* and `listener` is of type *DocumentListener*. The *Document* class also has methods that make it easy to read a document from a file and write a document to a file. I won't discuss all the things you can do with text components here. For one more peek at their capabilities, see the sample program *SimpleRTFEdit.java*, a very minimal editor for files that contain styled text of type "text/rtf."

### 13.4.5 Custom Components

Java's standard component classes are usually all you need to construct a user interface. At some point, however, you might need a component that Java doesn't provide. In that case, you can write your own component class, building on one of the components that Java does provide. We've already done this, actually, every time we've written a subclass of the *JPanel* class to use as a drawing surface. A *JPanel* is a blank slate. By defining a subclass, you can make it show any picture you like, and you can program it to respond in any way to mouse and keyboard events. Sometimes, if you are lucky, you don't need such freedom, and you can build on one of Java's more sophisticated component classes.

For example, suppose I have a need for a "stopwatch" component. When the user clicks on the stopwatch, I want it to start timing. When the user clicks again, I want it to display the elapsed time since the first click. The textual display can be done with a *JLabel*, but we want a *JLabel* that can respond to mouse clicks. We can get this behavior by defining a *StopWatchLabel* component as a subclass of the *JLabel* class. A *StopWatchLabel* object will listen for mouse clicks on itself. The first time the user clicks, it will change its display to "Timing..." and remember the time when the click occurred. When the user clicks again, it will check the time again, and it will compute and display the elapsed time. (Of course, I don't necessarily have to define a subclass. I could use a regular label in my program, set up a listener to respond to mouse events on the label, and let the program do the work of keeping track of the time and changing the text displayed on the label. However, by writing a new class, I have something that can be **reused** in other projects. I also have all the code involved in the stopwatch function collected together neatly in one place. For more complicated components, both of these considerations are very important.)

The *StopWatchLabel* class is not very hard to write. I need an instance variable to record the time when the user starts the stopwatch. Times in Java are measured in milliseconds and are stored in variables of type long (to allow for very large values). In the `mousePressed()` method, I need to know whether the timer is being started or stopped, so I need a boolean instance variable, `running`, to keep track of this aspect of the component's state. There is one more item of interest: How do I know what time the mouse was clicked? The method `System.currentTimeMillis()` returns the current time. But there can be some delay between the time the user clicks the mouse and the time when the `mousePressed()` routine is called. To make my stopwatch as accurate as possible, I don't want to know the current time. I want to know the exact time when the mouse was pressed. When I wrote the *StopWatchLabel* class, this need sent me on a search in the Java documentation. I found that if `evt` is an object of type *MouseEvent*, then the function `evt.getWhen()` returns the time when the event occurred. I call this function in the `mousePressed()` routine to determine the exact time when the user clicked on the label. The complete `StopWatch` class is rather short:

```
import java.awt.event.*;
import javax.swing.*;

/**
 * A custom component that acts as a simple stop-watch.  When the
user clicks
```

```
 * on it, this component starts timing.  When the user clicks
again,
 * it displays the time between the two clicks.  Clicking a third
time
 * starts another timer, etc.  While it is timing, the label just
 * displays the message "Timing....".
 */
public class StopWatchLabel extends JLabel implements MouseListener
{

   private long startTime;   // Start time of timer.
                             //   (Time is measured in
milliseconds.)

   private boolean running;  // True when the timer is running.

   /**
    * Constructor sets initial text on the label to
    * "Click to start timer." and sets up a mouse listener
    * so the label can respond to clicks.
    */
   public StopWatchLabel() {
      super("  Click to start timer.  ", JLabel.CENTER);
      addMouseListener(this);
   }


   /**
    * Tells whether the timer is currently running.
    */
   public boolean isRunning() {
      return running;
   }


   /**
    * React when the user presses the mouse by starting or stopping
    * the timer and changing the text that is shown on the label.
    */
   public void mousePressed(MouseEvent evt) {
      if (running == false) {
            // Record the time and start the timer.
         running = true;
         startTime = evt.getWhen();  // Time when mouse was
clicked.
         setText("Timing....");
      }
      else {
            // Stop the timer.  Compute the elapsed time since the
            // timer was started and display it.
         running = false;
         long endTime = evt.getWhen();
         double seconds = (endTime - startTime) / 1000.0;
         setText("Time: " + seconds + " sec.");
      }
   }
   public void mouseReleased(MouseEvent evt) { }
   public void mouseClicked(MouseEvent evt) { }
   public void mouseEntered(MouseEvent evt) { }
   public void mouseExited(MouseEvent evt) { }

}
```

Don't forget that since *StopWatchLabel* is a subclass of *JLabel*, you can do anything with a *StopWatchLabel* that you can do with a *JLabel*. You can add it to a container. You can set its font, foreground color, and background color. You can set the text that it displays (although this would interfere with its stopwatch function). You can even add a *Border* if you want.

Let's look at one more example of defining a custom component. Suppose that -- for no good reason whatsoever -- I want a component that acts like a *JLabel* except that it displays its text in mirror-reversed form. Since no standard component does anything like this, the *MirrorText* class is defined as a subclass of *JPanel*. It has a constructor that specifies the text to be displayed and a `setText()` method that changes the displayed text. The `paintComponent()` method draws the text mirror-reversed, in the center of the component. This uses techniques discussed in Subsection 13.1.1 and Subsection 13.2.1. Information from a *FontMetrics* object is used to center the text in the component. The reversal is achieved by using an off-screen canvas. The text is drawn to the off-screen canvas, in the usual way. Then the image is copied to the screen with the following command, where `OSC` is the variable that refers to the off-screen canvas, and `width` and `height` give the size of both the component and the off-screen canvas:

```
g.drawImage(OSC, width, 0, 0, height, 0, 0, width, height, null);
```

This is the version of `drawImage()` that specifies corners of destination and source rectangles. The corner `(0,0)` in `OSC` is matched to the corner `(width,0)` on the screen, while `(width,height)` is matched to `(0,height)`. This reverses the image left-to-right. Here is the complete class:

```
import java.awt.*;
import javax.swing.*;
import java.awt.image.BufferedImage;

/**
 * A component for displaying a mirror-reversed line of text.
 * The text will be centered in the available space.  This
component
 * is defined as a subclass of JPanel.  It respects any background
 * color, foreground color, and font that are set for the JPanel.
 * The setText(String) method can be used to change the displayed
 * text.  Changing the text will also call revalidate() on this
 * component.
 */
public class MirrorText extends JPanel {

   private String text; // The text displayed by this component.

   private BufferedImage OSC; // Holds an un-reversed picture of
the text.
```

```
   /**
    * Construct a MirrorText component that will display the
specified
    * text in mirror-reversed form.
    */
   public MirrorText(String text) {
      if (text == null)
         text = "";
      this.text = text;
   }

   /**
    * Change the text that is displayed on the label.
    * @param text the new text to display
    */
   public void setText(String text) {
      if (text == null)
         text = "";
      if ( ! text.equals(this.text) ) {
         this.text = text;  // Change the instance variable.
         revalidate();      // Tell container to recompute its
layout.
         repaint();         // Make sure component is redrawn.
      }
   }

   /**
    * Return the text that is displayed on this component.
    * The return value is non-null but can be an empty string.
    */
   public String getText() {
      return text;
   }

   /**
    * The paintComponent method makes a new off-screen canvas, if
necessary,
    * writes the text to the off-screen canvas, then copies the
canvas onto
    * the screen in mirror-reversed form.
    */
   public void paintComponent(Graphics g) {
      int width = getWidth();
      int height = getHeight();
      if (OSC == null || width != OSC.getWidth()
                        || height != OSC.getHeight()) {
         OSC = new
BufferedImage(width,height,BufferedImage.TYPE_INT_RGB);
      }
      Graphics OSG = OSC.getGraphics();
      OSG.setColor(getBackground());
      OSG.fillRect(0, 0, width, height);
      OSG.setColor(getForeground());
      OSG.setFont(getFont());
      FontMetrics fm = OSG.getFontMetrics(getFont());
      int x = (width - fm.stringWidth(text)) / 2;
      int y = (height + fm.getAscent() - fm.getDescent()) / 2;
      OSG.drawString(text, x, y);
      OSG.dispose();
      g.drawImage(OSC, width, 0, 0, height, 0, 0, width, height,
null);
   }

   /**
    * Compute a preferred size that includes the size of the text,
plus
    * a boundary of 5 pixels on each edge.
    */
   public Dimension getPreferredSize() {
      FontMetrics fm = getFontMetrics(getFont());
      return new Dimension(fm.stringWidth(text) + 10,
            fm.getAscent() + fm.getDescent() + 10);
   }

}  // end MirrorText
```

This class defines the method "`public Dimension getPreferredSize()`". This method is called by a layout manager when it wants to know how big the component would like to be. Standard components come with a way of computing a preferred size. For a custom component based on a *JPanel*, it's a good idea to provide a custom preferred size. Every component has a method `setPreferredSize()` that can be used to set the preferred size of the component. For our *MirrorText* component, however, the preferred size depends on the font and the text of the component, and these can change from time to time. We need a way to compute a preferred size on demand, based on the current font and text. That's what we do by defining a `getPreferredSize()` method. The system calls this method when it wants to know the preferred size of the component. In response, we can compute the preferred size based on the current font and text.

The *StopWatchLabel* and *MirrorText* classes define components. Components don't stand on their own. You have to add them to a panel or other container. The sample program *CustomComponentTest.java* demonstrates using a *MirrorText* and a *StopWatchLabel* component, which are defined by the source code files *MirrorText.java* and *StopWatchLabel.java*.

In this program, the two custom components and a button are added to a panel that uses a *FlowLayout* as its layout manager, so the components are not arranged very neatly. If you click the button labeled "Change Text in this Program", the text in all the components will be changed. You can also click on the stopwatch label to start and stop the stopwatch. When you do any of these things, you will notice that the components will be rearranged to take the new sizes into account. This is known as "validating" the container. This is done automatically when a standard component changes in some way that requires a change in preferred size or location. This may or may not be the behavior that you want. (Validation doesn't always cause as much disruption as it does in this program. For example, in a *GridLayout*, where all the components are displayed at the same size, it will have no effect at all. I chose a *FlowLayout* for this example to make the effect more obvious.) When the text is changed in a *MirrorText* component, there is no automatic validation of its container. A custom component such as *MirrorText* must call the `revalidate()` method to indicate that the container that contains the component should be validated. In the *MirrorText* class, `revalidate()` is called in the `setText()` method.

# Finishing Touches

IN THIS FINAL SECTION, I will present a program that is more complex and more polished than those we have looked at previously. Most of the examples in this book have been "toy" programs that illustrated one or two points about programming techniques. It's time to put it all together into a full-scale program that uses many of the techniques that we have covered, and a few more besides. After discussing the program and its basic design, I'll use it as an excuse to talk briefly about some of the features of Java that didn't fit into the rest of this book.

The program that we will look at is a Mandelbrot Viewer that lets the user explore the famous Mandelbrot set. I will begin by explaining what that means. Note that an even more capable Mandelbrot program can be found at http://math.hws.edu/eck/xJava/MB.

## 13.5.1 The Mandelbrot Set

The Mandelbrot set is a set of points in the xy-plane that is defined by a computational procedure. To use the program, all you really need to know is that the Mandelbrot set can be used to make some pretty pictures, but here are the mathematical details: Consider the point that has real-number coordinates `(a,b)` and apply the following computation:

```
Let x = a
Let y = b
Repeat:
   Let newX = x*x - y*y + a
   Let newY = 2*x*y + b
   Let x = newX
   Let y = newY
```

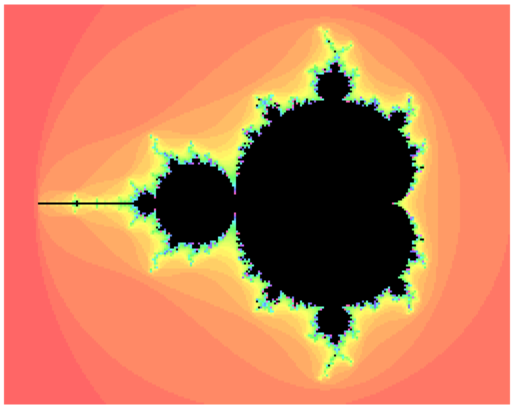
As the loop is repeated, the point `(x,y)` changes. The question for the Mandelbrot set is, does `(x,y)` grow without bound, or is it trapped forever in a finite region of the plane? If `(x,y)` escapes to infinity (that is, grows without bound), then the starting point `(a,b)` is **not** in the Mandelbrot set. If `(x,y)` is trapped in a finite region, then `(a,b)` is in the Mandelbrot set. Now, it is known that if $x^2 + y^2$ ever becomes strictly greater than 4, then `(x,y)` will escape to infinity. So, if $x^2 + y^2$ ever becomes bigger than 4 in the above loop, we can end the loop and say that `(a,b)` is definitely not in the Mandelbrot set. For a point `(a,b)` in the Mandelbrot set, the loop will never end. When we do this on a computer, of course, we don't want to have a loop that runs forever, so we put a limit on the number of times that the loop is executed:

```
x = a;
y = b;
```

```
count = 0;
while ( x*x + y*y < 4.1 ) {
   count++;
   if (count > maxIterations)
      break;
   double newX = x*x - y*y + a;
   double newY = 2*x*y + b;
   x = newY;
   y = newY;
}
```

After this loop ends, if `count` is less than or equal to `maxIterations`, we can say that `(a,b)` is definitely not in the Mandelbrot set. If `count` is greater than `maxIterations`, then `(a,b)` might or might not be in the Mandelbrot set, but the larger `maxIterations` is, the more likely that `(a,b)` is actually in the set.

To make a picture from this procedure, use a rectangular grid of pixels to represent some rectangle in the plane. Each pixel corresponds to some real number coordinates `(a,b)`. (Use the coordinates of the center of the pixel.) Run the above loop for each pixel. If the `count` goes past `maxIterations`, color the pixel black; this is a point that is *possibly* in the Mandelbrot set. Otherwise, base the color of the pixel on the value of `count` after the loop ends, using different colors for different counts. In some sense, the higher the count, the closer the point is to the Mandelbrot set, so the colors give some information about points outside the set and about the shape of the set. However, it's important to understand that the colors are arbitrary and that colored points are definitely **not** in the set. Here is a picture that was produced by the Mandelbrot Viewer program using this computation. The black region is the Mandelbrot set (except that not all black points are known to be definitely in the set):



When you use the program, you can "zoom in" on small regions of the plane. To do so, just click-and-drag the mouse on the picture. This will draw a rectangle around part of the picture. When you release the mouse, the part of the picture inside the rectangle will be zoomed to fill the entire display. If you simply click a point in the picture, you will zoom in on the point where you click by a magnification factor of two. (Shift-click or use the right mouse button to zoom out instead of zooming in.) The interesting points are along the boundary of the Mandelbrot set. In fact, the boundary is infinitely complex. (Note that if you zoom in too far, you will exceed the capabilities of the double data type; nothing is done in the program to prevent this.)

Use the "MaxIterations" menu to increase the maximum number of iterations in the loop. Remember that black pixels might or might not be in the set; when you increase "MaxIterations," you might find that a black region becomes filled with color. The "Palette" menu determines the set of colors that are used. Different palettes give very different visualizations of the set. The "PaletteLength" menu determines how many different colors are used. In the default setting, a different color is used for each possible value of `count` in the algorithm. Sometimes, you can get a much better picture by using a different number of colors. If the palette length is less than `maxIterations`, the palette is repeated to cover all the possible values of `count`; if the palette length is greater than `maxIterations`, only part of of the palette will be used. (If the picture is of an almost uniform color, try *decreasing* the palette length, since that makes the color vary more quickly as `count` changes. If you see what look like randomly colored dots instead of bands of color, try *increasing* the palette length.)

The program has a "File" menu that can be used to save the picture as a PNG image file. You can also save a "param" file which simply saves the settings that produced the current picture. A param file can be read back into the program using the "Open" command.

The Mandelbrot set is named after Benoit Mandelbrot, who was the first person to note the incredible complexity of the set. It is astonishing that such complexity and beauty can arise out of such a simple algorithm.

---

### 13.5.2 Design of the Program

Most classes in Java are defined in packages. While we have used standard packages such as `javax.swing` and `java.io` extensively, almost all of my programming examples have been in the "default package," which means that they are not declared to belong to any named package. However, when doing more serious programming, it is good style to create a package to hold the classes for your program. The Oracle corporation recommends that package names should be based on an Internet domain name of the organization that produces the package. My office computer has domain name `eck.hws.edu`, and no other computer in the world should have the same name. According to Oracle, this allows me to use the package name `edu.hws.eck`, with the elements of the domain name in reverse order. I can also use sub-packages of this package, such as `edu.hws.eck.mdb`, which is the package name that I decided to use for my Mandelbrot Viewer application. No one else -- or at least no one else who uses the same naming convention -- will ever use the same package name, so this package name uniquely identifies my program.

I briefly discussed using packages in Subsection 2.6.6 and in the context of the programming examples in Section 12.5 Here's what you need to know for the Mandelbrot Viewer program: The program is defined in eight Java source code files. They can be found in the directory *edu/hws/eck/mdb* inside the `source` directory of the web site. (That is, they are in a directory named `mdb`, which is inside a directory named `eck`, which is inside `hws`, which is inside `edu`. The directory structure must follow the package name in this way.) The same directory also contains a file named *strings.properties* that is used by the program and that will be discussed below. And there is an `examples` folder that contains resource files used for an "Examples" menu. For an Integrated Development Environment such as Eclipse, you should just have to add the `edu` directory to your project. To compile the files on the command line, you must be working in the directory that contains the `edu` directory. Use the command

```
javac  edu/hws/eck/mdb/*.java
```

or, if you use Windows,

```
javac  edu\hws\eck\mdb\*.java
```

to compile the source code. The main routine for the program is defined by a class named *Main*. To run this class, use the command:

```
java  edu.hws.eck.mdb.Main
```

This command must also be given in the directory that contains the `edu` directory.

---

The work of computing and displaying images of the Mandelbrot set is done in *MandelbrotDisplay.java*. The *MandelbrotDisplay* class is a subclass of *JPanel*. It uses an off-screen canvas to hold a copy of the image. (See Subsection 13.1.1.) The `paintComponent()` method copies this image onto the panel. Then, if the user is drawing a "zoom box" with the mouse, the zoom box is drawn on top of the image. In addition to the image, the class uses a two-dimensional array to store the iteration count for each pixel in the image. If the range of xy-values changes, or if the size of the window changes, all the counts must be recomputed. Since the computation can take quite a while, it would not be acceptable to block the user interface while the computation is being performed. The solution is to do the computation in separate "worker" threads, as discussed in Chapter 12. The program uses one worker thread for each available processor. When the computation begins, the image is filled with gray. Every so often, about twice a second, the data that has been computed by the computation threads is gathered and applied to the off-screen canvas, and the part of the canvas that has been modified is copied to the screen. A *Timer* is used to control this process -- each time the timer fires, the image is updated with any new data that has been computed by the threads. The user can continue to use the menus and even the mouse while the image is being computed.

The file *MandelbrotPanel.java* defines the main panel of the Mandelbrot Viewer window. *MandelbrotPanel* is another subclass of *JPanel*. A *MandelbrotPanel* is mostly filled with a *MandelbrotDisplay*. It also adds a *JLabel* beneath the display. The *JLabel* is used as a "status bar" that shows some information that might be interesting to the user. The *MandelbrotPanel* also defines the program's mouse listener. In addition to handling zooming, the mouse listener puts the x and y coordinates of the current mouse location in the status bar as the user moves or drags the mouse. Also, when the mouse exits the drawing area, the text in the status bar is set to read "Idle". This is the first time that we have seen an actual use for `mouseMoved` and `mouseExited` events. (See Subsection 6.3.2 and Subsection 6.3.4.)

The menu bar for the program is defined in *Menus.java*. Commands in the "File" and "Control" menu are defined as *Actions*. (See Subsection 13.3.1.) Note that among the actions are file manipulation commands that use techniques from Subsection 11.2.3, Subsection 11.5.2, and Subsection 13.1.5. The "MaxIterations," "Palette," and "PaletteLength" menus each contain a group of *JRadioButtonMenuItems*. (See Subsection 13.3.3.) I have tried several approaches for handling such groups, and none of them have satisfied me completely. In this program, I have defined a nested class inside *Menus* to represent each group. For example, the *PaletteManager* class contains the menu items in the "Palette" menu as instance variables. It registers an action listener with each item, and it defines a few utility routines for operating on the menu. The classes for the three menus are very similar and should probably have been defined as subclasses of some more general class. There is an "Examples" menu that contains settings for several sample views of pieces of the Mandelbrot set.

The *MandelbrotPanel* that is being used in the program is a parameter to the *Menus* constructor. Many of the menu commands operate on this panel or on the *MandelbrotDisplay* that it contains. In order to carry out these commands, the *Menus* object needs a reference to the *MandelbrotPanel*. As for the *MandelbrotDisplay*, the panel has a method `getDisplay()` that returns a reference to the display that it contains. So as long as the menu bar has a reference to the panel, it can obtain a reference to the display. In previous examples, everything was written as one large class file, so all the objects were directly available to all the code. When a program is made up of multiple interacting files, getting access to the necessary objects can be more of a problem.

*MandelbrotPanel*, *MandelbrotDisplay*, and *Menus* are the main classes that make up the Mandelbrot Viewer program. *MandelbrotFrame.java* defines a simple subclass of *JFrame* that shows a MandelbrotPanel and its menu bar. And *Main.java* contains the `main()` routine that actually runs the program. There are a few other classes that I will discuss below.

This brief discussion of the design of the Mandelbrot Viewer has shown that it uses a wide variety of techniques that were covered earlier in this book. In the rest of this section, we'll look at a few new features of Java that were used in the program.

---

### 13.5.3 Internationalization

Internationalization refers to writing a program that is easy to adapt for running in different parts of the world. Internationalization is often referred to as I18n, where 18 is the number of letters between the "I" and the final "n" in "Internationalization." The process of adapting the program to a particular location is called localization, and the locations are called locales. Locales differ in many ways, including the type of currency used and the format used for numbers and dates, but the most obvious difference is language. Here, I will discuss how to write a program so that it can be easily translated into other languages.

The key idea is that strings that will be presented to the user should not be coded into the program source code. If they were, then a translator would have to search through the entire source code, replacing every string with its translation. Then the program would have to be recompiled. In a properly internationalized program, all the strings are stored together in one or more files that are separate from the source code, where they can easily be found and translated. And since the source code doesn't have to be modified to do the translation, no recompilation is necessary.

To implement this idea, the strings are stored in one or more properties files. A properties file is just a list of key/value pairs. For translation purposes, the values are strings that will be presented to the user; these are the strings that have to be translated. The keys are also strings, but they don't have to be translated because they will never be presented to the user. Since they won't have to be modified, the key strings can be used in the program source code. Each key uniquely identifies one of the value strings. The program can use the key string to look up the corresponding value string from the properties file. The program only needs to know the key string; the user will only see the value string. When the properties file is translated, the user of the program will see different value strings.

The format of a properties file is very simple. The key/value pairs take the form

```
key.string=value string
```

There are no spaces in the key string or before the equals sign. Periods are often used to divide words in the key string. The value string can contain spaces or any other characters. If the line ends with a backslash ("\"), the value string is continued on the next line; in this case, spaces at the beginning of that line are ignored. One unfortunate detail is that a properties file can contain only plain ASCII characters. The ASCII character set only supports the English alphabet. Nevertheless, a value string can include arbitrary UNICODE characters. Non-ASCII characters just have to be specially encoded. The JDK comes with a program, *native2ascii*, that can convert files that use non-ASCII characters into a form that is suitable for use as a properties file.

Suppose that the program wants to present a string to the user (as the name of a menu command, for example). The properties file would contain a key/value pair such as

```
menu.saveimage=Save PNG Image...
```

where "Save PNG Image..." is the string that will appear in the menu. The program would use the key string, "menu.saveimage", to look up the corresponding value string and would then use the value string as the text of the menu item. In Java, the look up process is supported by the *ResourceBundle* class, which knows how to retrieve and use properties files. Sometimes a string that is presented to the user contains substrings that are not known until the time when the program is running. A typical example is the name of a file. Suppose, for example, that the program wants to tell the user, "Sorry, the file, **filename**, cannot be loaded", where **filename** is the name of a file that was selected by the user at run time. To handle cases like this, value strings in properties files can include placeholders that will be replaced by strings to be determined by the program at run time. The placeholders take the form "`{0}`", "`{1}`", "`{2}`", .... For the file error example, the properties file might contain:

```
error.cantLoad=Sorry, the file, {0}, cannot be loaded
```

The program would fetch the value string for the key `error.cantLoad`. It would then substitute the actual file name for the placeholder, "`{0}`". Note that when the string is translated, the word order might be completely different. By using a placeholder for the file name, you can be sure that the file name will be put in the correct grammatical position for the language that is being used. Placeholder substitution is not handled by the *ResourceBundle* class, but Java has another class, *MessageFormat*, that makes such substitutions easy.

For the Mandelbrot Viewer program, the properties file is *strings.properties*. (Any properties file should have a name that ends in "`.properties`".) Any string that you see when you run the program comes from this file. For handling value string lookup, I wrote *I18n.java*. The *I18n* class has a static method

```
        public static tr( String key, Object... args )
```

that handles the whole process. Here, `key` is the key string that will be looked up in `strings.properties`. Additional parameters, if any, will be substituted for placeholders in the value string. (Recall that the formal parameter declaration "`Object...`" means that there can be any number of actual parameters after `key`; see Subsection 7.1.2.) Typical uses would include:

```
        String saveImageCommandText = I18n.tr( "menu.saveimage" );

        String errMess = I18n.tr( "error.cantLoad" , selectedFile.getName()
        );
```

You will see function calls like this throughout the Mandelbrot Viewer source code. The *I18n* class is written in a general way so that it can be used in any program. As long as you provide a properties file as a resource, the only things you need to do are change the resource file name in `I18n.java` and put the class in your own package.

It is actually possible to provide several alternative properties files in the same program. For example, you might include French and Japanese versions of the properties file along with an English version. If the English properties file is named `strings.properties`, then the names for the French and Japanese versions should be `strings_fr.properties` and `strings_ja.properties`. Every language has a two-letter code, such as "fr" and "ja", that is used in constructing properties file names for that language. The program asks for the properties file using the simple name "`strings`". If the program is being run on a Java system in which the preferred language is French, the program will try to load "`strings_fr.properties`"; if that fails, it will look for "`strings.properties`". This means that the program will use the French properties files in a French locale; it will use the Japanese properties file in a Japanese locale; and in any other locale it will use the default properties file.

### 13.5.4 Events, Events, Events

We have worked extensively with mouse events, key events, and action events, but these are only a few of the event types that are used in Java. The Mandelbrot Viewer program makes use of several other types of events. It also serves as an example of the benefits of event-oriented programming.

Let's start from the following fact: The *MandelbrotDisplay* class knows nothing about any of the other classes that make up the program (with the single exception of one call to the internationalization method `I18n.tr`). Yet other classes are aware of things that are going on in the *MandelbrotDisplay* class. For example, when the size of the display is changed, the new size is reported in the status bar that is part of the *MandelbrotPanel* class. In the *Menus* class, certain menus are disabled when the display begins the computation of an image and are re-enabled when the computation completes. The display doesn't call methods in the *MandelbrotPanel* or *Menus* classes, so how do these classes get their information about what is going on in the display? The answer, of course, is events. The *MandelbrotDisplay* object emits events of various types when various things happen. The *MandelbrotPanel* and *Menus* objects set up listeners that hear those events and respond to them.

The point is that because events are used for communication, the *MandelbrotDisplay* class is not strongly coupled to the other classes. In fact, it can be used in other programs without any modification and without access to the other classes. The alternative to using events would be to have the display object call methods such as `displaySizeChanged()` or `computationStarted()` in the *MandelbrotPanel* and *MandelbrotFrame* objects to tell them what is going on in the display. This would be strong coupling: Any programmer who wanted to use *MandelbrotDisplay* would also have to use the other two classes or would have to modify the display class so that it no longer refers to the other classes. Of course, not everything can be done with events and not all strong coupling is bad: The *MandelbrotPanel* class refers directly to the *MandelbrotDisplay* class and cannot be used without it -- but since the whole purpose of a *MandelbrotPanel* is to hold a *MandelbrotDisplay*, the coupling is not a problem.

---

The Mandelbrot Viewer program responds to mouse events on the display. These events are generated by the display object, but the display class itself doesn't care about mouse events and doesn't do anything in response to them. Mouse events are handled by a listener in the *MandelbrotPanel*, which responds to them by zooming the display and by showing mouse coordinates in the status bar.

The status bar also shows the new size of the display whenever that size is changed. To handle this, events of type *ComponentEvent* are used. When the size of a component is changed, a *ComponentEvent* is generated. In the Mandelbrot Viewer program, a *ComponentListener* in the *MandelbrotPanel* class listens for size-change events in the display. When one occurs, the listener responds by showing the new size in the status bar; the display knows nothing about the status bar that shows the display's size.

Component events are also used internally in the *MandelbrotDisplay* class in an interesting way. When the user dynamically changes the size of the display, its size can change several times each second. Normally, a change of display size would trigger the creation of a new off-screen canvas and the start of a new asynchronous computation of the image. However, doing this is a big deal, not something I want to do several times in a second. If you try resizing the program's window, you'll notice that the image doesn't change size dynamically as the window size changes. The same image and off-screen canvas are used as long as the size is changing. Only about one-third of a second after the size has stopped changing will a new, resized image be produced. Here is how this works: The display sets up a *ComponentListener* to listen for resize events on itself. When a resize occurs, the listener starts a *Timer* that has a delay of 1/3 second. (See Subsection 6.4.1.) While this timer is running, the `paintComponent()` method does not resize the image; instead, it reuses the image that already exists. If the timer fires 1/3 second later, the image will be resized at that time. However, if another resize event occurs while the first timer is running, then the first timer will be stopped before it has a chance to fire, and a new timer will be started with a delay of 1/3 second. The result is that the image does not get resized until 1/3 second after the size of the window stops changing.

The Mandelbrot Viewer program also uses events of type *WindowEvent*, which are generated by a window when it opens or closes (among other things). Window events are used by *Main.java* to trigger an action that has to be taken when the program is ending; this will be discussed below.

Perhaps the most interesting use of events in the Mandelbrot Viewer program is to enable and disable menu commands based on the status of the display. For this, events of type *PropertyChangeEvent* are used. This event class is part of a "bean" framework that Java uses for some advanced work with objects, and class *PropertyChangeEvent* and related classes are defined in the package `java.beans`. The idea is that bean objects are defined by their "properties" (which are just aspects of the state of the bean). When a bean property changes, the bean can emit a *PropertyChangeEvent* to notify other objects of the change. Properties for which property change events are emitted are known technically as bound properties. A bound property has a **name** that identifies that particular property among all the properties of the bean. When a property change event is generated, the event object includes the name of the property that has changed, the previous value of the property, and the new value of the property.

The *MandelbrotDisplay* class has a bound property whose name is given by the constant `MandelbrotDisplay.STATUS_PROPERTY`. A display emits a property change event when its status changes. The possible values of the status property are given by other constants, such as `MandelbrotDisplay.STATUS_READY`. The `READY` status indicates that the display is not currently running a computation and is ready to do another one. There are several menu commands that should be enabled only when the status of the display is `READY`. To implement this, the *Menus* class defines a *PropertyChangeListener* to listen for property change events from the display. When this listener hears an event, it responds by enabling or disabling menu commands according to the new value of the status property.

All of Java's GUI components are beans and are capable of emitting property change events. In any subclass of *Component*, this can be done simply by calling the method

```
public void firePropertyChange(String propertyName,
                                        Object oldValue, Object
newValue)
```

For example, the *MandelbrotDisplay* class uses the following method for setting its current status:

```
private void setStatus(String status) {
   if (status == this.status) {
           // Note: Event should be fired only if status actually
changes.
      return;
   }
   String oldStatus = this.status;
   this.status = status;
   firePropertyChange(STATUS_PROPERTY, oldStatus, status);
}
```

When writing bean classes from scratch, you have to add support for property change events, if you need them. To make this easier, the `java.beans` package provides the *PropertyChangeSupport* class.

---

### 13.5.5 Custom Dialogs

Java has several standard dialog boxes that are defined in the classes *JOptionPane*, *JColorChooser*, and *JFileChooser*. These were introduced in Subsection 6.7.2 and Subsection 11.2.3. Dialogs of all these types are used in the Mandelbrot Viewer program. However, sometimes other types of dialog are needed. In such cases, you can build a custom dialog box.

Dialog boxes are defined by subclasses of the class *JDialog*. Like frames, dialog boxes are separate windows on the screen, and the *JDialog* class is very similar to the *JFrame* class. The big difference is that a dialog box has a parent, which is a frame or another dialog box that "owns" the dialog box. If the parent of a dialog box closes, the dialog box closes automatically. Furthermore, the dialog box might "float" on top of its parent, even when its parent is the active window.

Dialog boxes can be either modal or modeless. When a modal dialog is put up on the screen, the rest of the application is blocked until the dialog box is dismissed. This is the most common case, and all the standard dialog boxes are modal. Modeless dialog boxes are more like independent windows, since they can stay on the screen while the user interacts with other windows. There are no modeless dialogs in the Mandelbrot Viewer program.

The Mandelbrot Viewer program uses two custom dialog boxes. They are used to implement the "Set Image Size" and "Set Limits" commands and are defined by the files *SetImageSizeDialog.java* and *SetLimitsDialog.java*. The "set image size" dialog lets the user enter a new width and height for the Mandelbrot image. The "set limits" dialog lets the user input the minimum and maximum values for x and y that are shown in the image. The two dialog classes are very similar. In both classes, several *JTextFields* are used for user input. Two buttons named "OK" and "Cancel" are added to the window, and listeners are set up for these buttons. If the user clicks "OK", the listener checks whether the inputs in the text fields are legal; if not, an error message is displayed to the user and the dialog stays on the screen. If the input is legal when the user clicks "OK", the dialog is disposed. The dialog is also disposed if the user clicks "Cancel" or clicks the dialog box's close box. The net effect is that the dialog box stays on the screen until the user either cancels the dialog or enters legal values for the inputs and clicks "OK". The program can find out which of these occurred by calling a method named `getInput()` in the dialog object after showing the dialog. This method returns `null` if the dialog was canceled; otherwise it returns the user input.

To make my custom dialog boxes easy to use, I added a `static showDialog()` method to each dialog class. When this function is called, it shows the dialog, waits for it to be dismissed, and then returns the value of the `getInput()` method. This makes it possible to use my custom dialog boxes in much the same way as Java's standard dialog boxes are used.

Custom dialog boxes are not difficult to create and to use, if you already know about frames. I will not discuss them further here, but you can look at the source code file *SetImageSizeDialog.java* as a model.

---

### 13.5.6 Preferences

Most serious programs allow the user to set preferences. A preference is really just a piece of the program's state that is saved between runs of the program. In order to make preferences persistent from one run of the program to the next, the preferences could simply be saved to a file in the user's home directory. However, there would then be the problem of locating the file. There would be the problem of naming the file in a way that avoids conflicts with file names used by other programs. And there would be the problem of cluttering up the user's home directory with files that the user shouldn't even have to know about.

To deal with these problems, Java has a standard means of handling preferences. It is defined by the package `java.util.prefs`. In general, the only thing that you need from this package is the class named *Preferences*.

In the Mandelbrot Viewer program, the file *Main.java* has an example of using *Preferences*. `Main.java` contains the `main()` routine for the program.

In most programs, the user sets preferences in a custom dialog box. However, the Mandelbrot program doesn't have any preferences that are appropriate for that type of treatment. Instead, as an example, I automatically save a few aspects of the program's state as preferences. Every time the program starts up, it reads the preferences, if any are available. Every time the program terminates, it saves the preferences. (Saving the preferences poses an interesting problem because the program ends when the *MandelbrotFrame* window closes, not when the `main()`

routine ends. In fact, the `main()` routine ends as soon as the window appears on the screen. So, it isn't possible to save the preferences at the end of the main program. The solution is to use events: A listener listens for *WindowEvents* from the frame. When a window-closed event is received, indicating that the program is ending, the listener saves the preferences.)

Preferences for Java programs are stored in some platform-dependent form in some platform-dependent location. As a Java programmer, you don't have to worry about it; the Java preferences system knows where to store the data. There is still the problem of identifying the preferences for one program among all the possible Java programs that might be running on a computer. Java solves this problem in the same way that it solves the package naming problem. In fact, by convention, the preferences for a program are identified by the package name of the program, with a slight change in notation. For example, the Mandelbrot Viewer program is defined in the package `edu.hws.eck.mdb`, and its preferences are identified by the string "/edu/hws/eck/mdb". (The periods have been changed to "`/`", and an extra "`/`" has been added at the beginning.)

The preferences for a program are stored in something called a "node." The user preferences node for a given program identifier can be accessed as follows:

```
Preferences root = Preferences.userRoot();
Preferences node = root.node(pathName);
```

where `pathname` is the string, such as "/edu/hws/eck/mdb", that identifies the node. The node itself consists of a simple list of key/value pairs, where both the key and the value are strings. You can store any strings you want in preferences nodes -- they are really just a way of storing some persistent data between program runs. In general, though, the key string identifies some particular preference item, and the associated value string is the value of that preference. A *Preferences* object, `prefnode`, contains methods `prefnode.get(key)` for retrieving the value string associated with a given key and `prefnode.put(key,value)` for setting the value string for a given key.

In `Main.java`, I use preferences to store the shape and position of the program's window. This makes the size and shape of the window persistent between runs of the program; when you run the program, the window will be right where you left it the last time you ran it. I also store the name of the directory that is currently selected in the file dialog box that is used by the program for the Save and Open commands. This is particularly satisfying, since the default behavior for a file dialog box is to start in the user's home directory, which is hardly ever the place where the user wants to keep a program's files. With the preferences feature, I can switch to the right directory the first time I use the program, and from then on I'll automatically be back in that directory when I use the program again. You can look at the source code in *Main.java* for the details.

---

And that's it.... There's a lot more that I could say about Java and about programming in general, but this book is only "An Introduction to Programming Using Java," and it's time for our journey to end. I hope that it has been a pleasant journey for you, and I hope that I have helped you establish a foundation that you can use as a basis for further exploration.