Chapter 11

Tables of Content

Collections of information can be structured in many ways. We sometimes organize information into lists including to-do lists, guest lists, and best-seller lists. Tables, including multiplication tables and periodic tables, have been used to organize information since long before we had spreadsheets to help produce them. Occasionally we organize our relatives into family trees, our friends into phone trees, and our employees into organizational charts (which are really just trees).

Given that there are many ways of organizing the contents of a collection, Java provides several mechanisms for representing and manipulating collections. In Chapter 10, we saw how recursive structures could be used to manage collections. In this chapter we will introduce another feature of the Java language that supports collections of data, the array. In particular, we will see how arrays can be used to organize collections of data as lists and tables.

To introduce the use of arrays in a context that is both important and familiar, we will begin by focusing on just one application of arrays in this chapter. We will explore how arrays can be used to manipulate the information that describes a digital image. Even relatively small digital images are composed of thousands of dots of color known as *pixels*. We will see that arrays provide a very natural way to manipulate the collection of information that describes an image's pixels. We will learn how we can transform images by manipulating the arrays that describe them. We will explore array algorithms to perform operations like scaling and rotation of images.

11.1 A Picture is Worth 754 Words

When an image is represented within a computer, the colors of the parts of the image are represented numerically. This is accomplished by dividing the picture into a grid of tiny squares called pixels and then using numbers to describe the color of each pixel. There are many schemes that can be used to represent colors numerically. These schemes typically use several numbers to describe each color but differ in how they interpret these numbers. In one scheme, known as RGB, the numbers describe the amount of red, green, and blue light that should be mixed to produce a pixel's color. In another called HSB, three numbers are used to describe qualities of the color referred to as its hue, saturation and brightness. Of course, there is nothing magic about the number three. There is at least one system for describing colors that uses four values. This scheme is known as CYMK.

Things are a bit simpler for images containing only shades of gray like the photo in Figure 11.1. Being lovers of simplicity, we will initially limit our attention to such grayscale images.

In a grayscale image, a single number can be used to describe the brightness of each pixel. Small

Figure 11.1: Asa Gray (1810-1888), Fisher Professor of Natural History, Harvard University

numbers are typically used to describe dark pixels and large number are used to describe brighter pixels. Different schemes use different ranges of values for the darkest and brightest pixels. A very common approach is to limit the range of values so that each pixel's description fits in a single 8-bit unit of computer memory. In this case, a black pixel is encoded using 0 and the largest number that can be encoded in 8 bits, 255, is used to describe white pixels. Various shades of gray are associated with the values between 0 and 255. A dark gray pixel might have a brightness value of 80 and a light gray pixel's brightness might be 200.

The number of pixels per inch affects the quality of the image's representation. If the number is too small, the individual pixels will become visible producing an image that looks grainy. Computer screens typically display images using about 75 pixels per inch. Printed images usually require more.

The image in Figure 11.1 is represented using 110 pixels per inch. In total, it is 215 pixels wide and 300 pixels high for a total of 64,500 pixels. This makes it a bit difficult to look at every single pixel in the image. On the other hand, we can look at the individual pixels and the values used to represent them if we focus on just a small region within the image. For example, the image below shows just the right eye from Figure 11.1.

This image is just 19 pixels wide and 10 pixels high. It is represented using 190 pixel values. We can see this in Figure 11.2 which shows the same image with each pixel enlarged 16 times. If this is not clear, just step back and look at the image from a distance. As you move away from the page the squares of gray will blend into an image of an eye.

We can show how numeric values are used to represent the brightness values in the image of the eye by overlaying a table of the numeric values of the pixel brightnesses with an enlarged version of the pixels themselves as shown in Figure 11.3. In this figure, you can clearly see that values close to 255 are used to describe the brightest pixels at the edges of the image. On the other hand, the darkest pixel in this image is described by the numeric value 51. Of course, inside the computer's

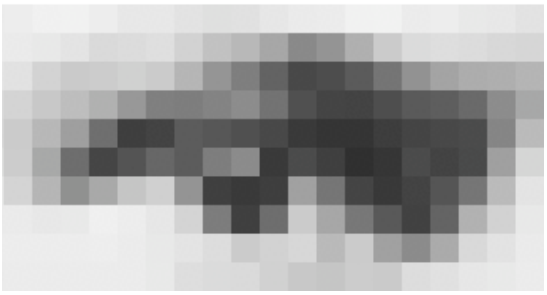

Figure 11.2: The right eye from Figure 11.1 enlarged by a factor of 16

237	240	241	237	231	231	226	223	224	230	234	235	241	242	235	237	233	231	226
234	232	227	217	219	222	209	193	183	165	137	147	174	201	218	221	219	215	213
225	210	200	203	207	202	180	148	126	99	75	78	92	117	145	162	171	173	179
211	199	188	176	151	127	125	131	139	115	81	71	69	79	90	95	106	139	166
201	182	158	111	65	69	93	87	81	75	60	56	56	64	70	75	77	133	186
203	167	110	72	84	103	93	125	138	61	77	66	51	57	77	69	76	159	212
211	180	143	168	196	206	147	66	61	65	160	118	70	59	69	85	118	186	227
234	231	233	239	236	230	211	124	66	110	201	165	119	89	68	100	177	209	232
236	236	236	238	235	232	227	220	202	209	214	185	201	160	138	170	216	226	234
235	235	235	235	235	233	221	218	217	208	199	196	203	200	220	224	230	233	235

Figure 11.3: The numbers describing pixel brightnesses for the right eye from Figure 11.1

memory, all that is really available is the collection of numeric values as shown in Figure 11.4.[1]

11.2 Matrices, Vectors, and Arrays

To write programs that work with images, we clearly need convenient ways to manipulate the large collections of numbers that describe pixels. In chapter 10 we saw that it is possible to define recursive classes to manage collections. All the collections we described using recursive classes, however, were viewed as lists. We do not think of the collection of numbers that describes an image as a list. As suggested by Figure 11.4, we think of these collections as tables. It would be best if we could manipulate them as tables within Java programs.

Java and most other programming languages include a mechanism called arrays that makes it possible to work with collections as either tables or lists. The notation used to work with arrays is borrowed from the notations that mathematicians have used for years when discussing matrices and vectors. If you look through a few linear algebra textbooks you will probably find a definition that reads something like:

[1]The 754 in the title of this section is the number of words required to write the contents of this table out in the form "two hundred and thirty seven, two hundred and forty, two hundred and forty one, two hundred and thirty seven, ..." If you take the time to check this count, please let me know if it is wrong.

237	240	241	237	231	231	226	223	224	230	234	235	241	242	235	237	233	231	226
234	232	227	217	219	222	209	193	183	165	137	147	174	201	218	221	219	215	213
225	210	200	203	207	202	180	148	126	99	75	78	92	117	145	162	171	173	179
211	199	188	176	151	127	125	131	139	115	81	71	69	79	90	95	106	139	166
201	182	158	111	65	69	93	87	81	75	60	56	56	64	70	75	77	133	186
203	167	110	72	84	103	93	125	138	61	77	66	51	57	77	69	76	159	212
211	180	143	168	196	206	147	66	61	65	160	118	70	59	69	85	118	186	227
234	231	233	239	236	230	211	124	66	110	201	165	119	89	68	100	177	209	232
236	236	236	238	235	232	227	220	202	209	214	185	201	160	138	170	216	226	234
235	235	235	235	235	233	221	218	217	208	199	196	203	200	220	224	230	233	235

Figure 11.4: The numbers describing pixel brightnesses for the right eye from Figure 11.1

Definition 1 *A rectangular collection of $m \times n$ values of the form*

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{bmatrix}$$

is called an m by n matrix.

When working with such a matrix, we use the name of the matrix, A, by itself to refer to the entire collection and we use the name together with two subscript values, such as $A_{3,5}$ to refer to a particular element of the collection.

Similarly, with Java arrays, we can use the name of an array itself, `pixels` for example, to refer to the entire collection, and the name together with subscript values to select a particular entry. In Java, however, the values that specify the position of the desired element are not actually written as subscripts. Instead, the values are placed in square brackets after the name of the array. For example, we could write

```
pixels[3][5] = 255;
```

to make the brightness value associated with the pixel at position 3,5 equal to the number that represents white. Since they are not actually positioned as subscripts, we often refer to the values in square brackets as *indices* rather than subscripts.

An array name followed by appropriate indices is called a *subscripted* or *indexed variable*. Note that a subscripted variable behaves just like a simple variable. When used within an expression, it represents the value currently associated with the specified position within an array. When placed on the left side of an assignment statement, it tells the computer to change the value associated with the specified position within an array. For example, the statement

```
pixels[3][5] = pixels[3][5] + 1;
```

might be executed if we wanted to make the pixel at position 3,5 in an image just a tiny bit brighter.

The values used to specify the indices in subscripted variables do not have to be literals. Just as we can say $A_{m,n}$ as well as $A_{3,5}$ in mathematical notation, we can say

```
pixels[m][n] = pixels[m][n] + 1;
```

to make the pixel at position m,n a little brighter. In fact, we can use arbitrarily complex expressions to describe index values. Something like

```
pixels[m - n][ 2*n + 1] = pixels[(m + n)/2][2*n] + 1;
```

is perfectly acceptable as long as the values described by the subscript expressions fall within the range of row and column numbers appropriate for the table named `pixels`.

Java's conventions for numbering the elements of an array are slightly different from those used by mathematicians working with matrices. Java starts counting at 0. That is, the top row is row 0 and the leftmost column is column 0. Also, when dealing with images in Java, the first index indicates the column number and the second index indicates the row number. This corresponds to the usual ordering mathematicians use for the x and y coordinates of a point in a graph, but is the reverse of the usual ordering mathematicians use for row and column numbers in a matrix. Given these differences, if we want to depict the elements of the array `pixels` in a manner similar to that used for A in Definition 1 we would use the layout:

$$\texttt{pixels} = \begin{bmatrix} \texttt{pixels}[0][0] & \texttt{pixels}[1][0] & \dots & \texttt{pixels}[m-1][0] \\ \texttt{pixels}[0][1] & \texttt{pixels}[1][1] & \dots & \texttt{pixels}[m-1][1] \\ \vdots & \vdots & \ddots & \vdots \\ \texttt{pixels}[0][n-1] & \texttt{pixels}[1][n-1] & \dots & \texttt{pixels}[m-1][n-1] \end{bmatrix}$$

11.3 Array Declarations

Recall that Java requires that all names used to refer to values or objects in a program be declared as local variables, instance variables or formal parameters. Declarations are formed by placing the name we wish to use after a description of the type of value that will be associated with the name. Depending on the situation, we may add modifiers like `private` and `final` or assign an initial value to a variable in its delcaration. Examples include

```
String response;
```

and

```
private JTextArea log = new JTextArea( WIDTH, HEIGHT );
```

Java provides a special notation to distinguish declarations of names that refer to arrays from declarations of names that refer to single values. If we declared the variable `pixels` as

```
int pixels;
```

Java would expect us to associate the name `pixels` with just a single `int`. To declare `pixels` as a name that will refer to a table of values we must instead write

```
int [ ] [ ] pixels;
```

This declaration illustrates the two kinds of information we must provide when declaring an array. The word `int` tells Java the type of values that will make up the collection associated with the array's name. The square brackets between `int` and `pixels` indicate that `pixels` will be the name of an array.

We place two pairs of brackets between `int` and `pixels` to indicate that two index values will be used to identify each element of the array `pixels`. In Java, it is possible to declare an array that uses more than two index values or just a single index value to identify specific elements. For example, in mathematics it is common to describe polynomials of high degree by placing subscripts on each term's coefficients. That is, we might write

$$c_0 \times x^0 + c_1 \times x^1 + c_2 \times x^2 + \cdots + c_{n-1} \times x^{n-1} + c_n \times x^n$$

and talk about the sequence of coefficients

$$< c_0, c_1, c_2, \ldots, c_n >$$

In Java, we would declare an array variable to hold the coefficient values using the declaration

```
int [] coefficients;
```

We say that the number of pairs of square brackets included in an array declaration determines the *dimensionality* of the array. Arrays that require only one index value are called one dimensional arrays, vectors, or sequences. Arrays that use two index values are called two dimensional arrays, matrices, or tables. It is also possible to have three dimensional arrays, four dimensional arrays, and so on.

Java arrays can also be used to manage collections of types other than `int`. For example, recall the calculator example presented in Chapter 7. That program displayed the buttons that formed its interface in a tabular pattern as shown in Figure 11.5. It might be useful to gather all of these buttons into an array named `keyboardButtons` in such a way that each `JButton` in the window could be accessed by specifying its row and column positions. That is,

```
keyboardButtons[2,2]
```

would refer to the "9" key. In this case, a declaration of the form

```
JButton [ ] [ ] keyboardButtons;
```

could be used to declare the variable `keyboardButtons`.

11.4 Array Constructions

Declaring a variable tells Java that we plan to use a name, but it does not tell Java what value or object should be associated with that name. For example, if we declare

```
private JButton addButton;
```

and the first statement executed in our program that used `addButton` was

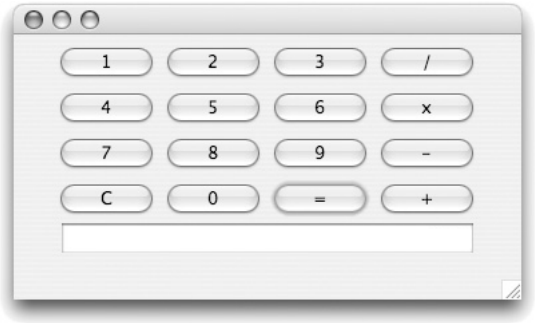

Figure 11.5: Calculator interface in which the keys are arranged in a table

```
contentPane.add( addButton );
```

the program would fail with an error because `addButton` has no value associated with it (or equivalently, it is associated with the value `null`).

To avoid such an error, we must include a construction that creates a button either in an assignment such as

```
addButton = new JButton( "+" );
```

or as an initial value specification in a revised declaration of the form

```
private JTextField addButton = new JButton( "+" );
```

Array names work in very similar ways. Declaring an array variable such as

```
JButton [ ] [ ] keyboardButtons;
```

tells the computer that we plan to use `keyboardButtons` to refer to an array of `JButton`s, but it does not tell the computer what collection of buttons the name should refer to. It does not even tell the computer how big the collection will be. If the name `keyboardButtons` is used in the calculator program shown in Figure 11.5, then the name should be associated with a 4 by 4 table of buttons. If it is instead used in a program with an interface resembling a standard telephone keypad, it should be associated with a 4 by 3 table of buttons. The computer cannot guess which sort of table we want to work with. We have to tell it. We can do this by constructing a new array and then assigning it to the variable name.

Array constructions looks like other constructions except that the parenthesized parameter list that is included in other constructions is replaced by a series of one or more integer values placed in square brackets. For example, to construct an array to hold the calculator buttons, we might use an array construction of the form

```
new JButton[4][4]
```

It is common to include array constructions as initial value specifications in array variable declarations such as

```
JButton [ ] [ ] keyboardButtons = new JButton[4][4];
```

Note that an array construction creates an array and nothing else. In particular, the construction

```
new JButton[4][4]
```

does not create any buttons. We have to create the desired buttons separately and associate them with the appropriate elements of the array.[2] This can be done using assignments like

```
keyboardButtons[0][1] = new JButton( "4" );
```

and

```
keyboardButtons[3][3] = new JButton( "+" );
```

The numbers placed in the square brackets in an array construction determine the range of possible index values that can be used to select specific elements of an array. Each number determines the number of distinct values that can be used for one of the index values required. For example, the declaration shown above would associate `keyboardButtons` with an array whose elements would be selected using two index values each falling between 0 and 3. Thus, there are four possible index values: 0, 1, 2 and 3. The number 4, however, would not be acceptable as an index value.

A declaration of the form

```
JButton [ ] [ ] keyboardButtons = new JButton[3][4];
```

would associate `keyboardButtons` with an array more appropriate for holding the buttons of a telephone keypad than a calculator. With this construction, the first index value used to select an element of the array would have to be 0, 1, or 2 while the second could range from 0 to 3. If `keyboardButtons` referred to the array created by this construction, the assignment

```
keyboardButtons[3][0] = new JButton( "3" );    // Warning:  Incorrect code!
```

would cause a program error identified as an "array index out of bounds exception". The correct assignment to place the "3" key in the upper right corner of the table would be

```
keyboardButtons[2][0] = new JButton( "3" );
```

In general, if we construct a matrix by saying

```
new SomeType[WIDTH][HEIGHT]
```

[2]In associating index values with buttons in the calculator and telephone keypad examples, we are using the same conventions we used for the table of pixel values. The first value in a pair of index values indicates horizontal placement while the second determines vertical position. There is nothing in the Java language that forces us to use this convention. We could equally well have switched the order of the index values.

According to standard mathematical conventions, when working with matrices the number indicating a value's vertical position comes first and the number for its horizontal position comes second. On the other hand, when using Cartesian coordinates, mathematicians place the number indicating horizontal placement first and the vertical position second. Java's scheme for identifying the positions of pixel values in an image is based on the conventions associated with Cartesian coordinates. Given the inconsistent conventions used in mathematical notation, we could certainly justify using the matrix-like convention for our keyboard tables. It is our hope, however, that by consistently specifying the horizontal position first and the vertical position second in all examples, we will minimize confusion.

the first index values used with the array will range from 0 to *WIDTH - 1*, and the second index values will range from 0 to *HEIGHT - 1*. The total number of entries in the matrix will be *WIDTH* $\times$ *HEIGHT*.

We have seen that one distinction between primitive types like `int` and `boolean` and classes like `JButton` is that we cannot construct values of primitive types. That is, we cannot say

```
new int( ... )
```

We can, however, construct arrays whose elements will be associated with values of primitive types. For example, if our program includes the declaration

```
int [ ] [ ] pixels;
```

we could execute the assignment

```
pixels = new int[19][10];
```

to associate `pixels` with a table of `int` values just the right size to hold the pixel values shown in Figure 11.4.

When we create an array of `JButton`s, Java does not create new `JButton`s to use as the elements of the array. We have to write separate instructions to create buttons and associate them with the array elements. Similarly, we have to write separate instructions to associate the correct `int` values with the elements of an array like `pixels`. Initially, the computer will simply associate the value 0 with all elements of the array. Therefore, the values found in the array `pixels` after it is initially constructed would describe the image

rather than

11.5 SImages and JLabels

While we know that a matrix full of zeroes can be interpreted as the description of a black rectangle, an array like `pixels` is not an image. It is a collection of numbers, not a collection of colored dots on the screen. We need some way to turn an array of numbers into an image we can see on a computer screen. This ability is provided through a Java class named`SImage`.[3]

There are several ways to construct an `SImage`. The constructor we present here is designed for creating grayscale images. If `pixels` is a two dimensional table of `int`s, then a construction of the form

```
new SImage( pixels )
```

[3]I bet you are wondering whether the "S" in `SImage` stands for "Sharper" or "Self". Actually, it stands for Squint. Like `NetConnection`, `SImage` is a part of the Squint library designed for use with this text. Within the standard Java libraries, the functionality provided by `SImage` is supported through a class named `BufferedImage`. `SImage` is designed to make the features of the `BufferedImage` class a bit more accessible.

will create a grayscale image whose pixels' brightnesses correspond to the values in the elements of the array. As explained above, the values in the array used in such a construction should fall between 0 and 255, but the `SImage` class is forgiving. Any value in the array that is less than 0 will be treated as if it was replaced with its absolute value, and any value greater than 255 will be treated as 255.

Constructing an `SImage` creates a new representation of the image in the computer's memory, but it does not immediately make the image appear on the computer's screen. In this way, `SImage`s are a bit like GUI components. For example, evaluating the construction

```
new JButton( "Click Me" )
```

immediately creates an object that represents a button in the computer's memory, but we have to do more to make the button appear on the screen. In the case of a GUI component, we have to add the component to the `contentPane` to make it appear.

`SImage`s are not GUI components. We cannot add them to the `contentPane`. Instead, displaying an `SImage` is more like displaying a `String`. We cannot add a `String` to the `contentPane`, but we can use the `setText` method to tell a `JButton` or `JLabel` to display the `String` as its contents. In addition to the `setText` method, the `JLabel` and `JButton` classes provide a method named `setIcon`. If we pass an `SImage` as a parameter in an invocation of this method, the `JLabel` or `JButton` on which the method is invoked will display the image represented by the `SImage`.

Figure 11.6 shows a very simple program that uses this feature to display our 19 by 10 rectangle of black pixels. The figure also includes a picture of the display the program would produce. The program creates a `JLabel` named `imageDisplay` and adds it to the `contentPane`. It then creates an `SImage` from a 19 by 10 table of `int`s. Finally, it uses the `setIcon` method to place this `SImage` in the `JLabel`.

Of course, the little black rectangle displayed in Figure 11.6 is not the most interesting image around. We could produce a slightly more interesting image by changing some of the pixel brightness values from 0 to larger numbers. For example, if we add the instructions

```
pixels[7][4] = 150;
pixels[8][5] = 200;
pixels[9][4] = 255;
pixels[9][5] = 255;
pixels[10][4] = 200;
pixels[11][5] = 150;
```

to our program, the image displayed will look like:

(OK! I admit it is a bit hard to see, but with just a bit of imagination, you will realize that this image looks just like a spiral galaxy floating in the blackness of space.) What we would really like to do, however, is take a picture from an image file created using a digital camera or scanner and access its pixel values within a Java program.

11.6 Image Files

The data in an image file can be converted into an `SImage` using a second form of constructor associated with the class. To use this constructor, the programmer passes a `String` containing the

```
import javax.swing.*;
import squint.*;

public class RectangleViewer extends GUIManager {
    private final int WINDOW_WIDTH = 100, WINDOW_HEIGHT = 100;

    private JLabel imageDisplay = new JLabel();

    public ImageViewer() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
        contentPane.add( imageDisplay );

        int [][] pixels = new int[19][10];
        SImage blackBox = new SImage( pixels );
        imageDisplay.setIcon( blackBox );
    }
}
```

Figure 11.6: A very simple program that displays an `SImage`

name of the image file as a parameter. For example, if the picture shown in Figure 11.1 is stored in the file `asaGray.jpg`, then the construction

```
new SImage( "asaGray.jpg" )
```

will produce an `Simage` representing this photograph. Then, this `SImage` can be used as the icon of a `JLabel` to make the picture appear on the screen.

Using this form of the `SImage` constructor, it is possible to write a program that can load an image and then modify its pixel values in various ways such as resizing, cropping, or brightening. That is, we can construct image processing software. When writing such a program, however, we would not want to specify the file to use by typing a literal like `"asaGray.jpg"` into the program's code. Instead, we would like to make it possible for the program's user to select the file to use through a mechanism like the dialog box shown in Figure 11.7. To make this easy, the Java Swing library contains a class called `JFileChooser`.

A `JFileChooser` is a GUI component, but unlike the components we have seen previously, a `JFileChooser` does not become part of the existing program window. Instead, it pops up a new window like the one shown in Figure 11.7.

The first step in using a `JFileChooser` is to construct one. The constructor is usually included in a local variable declaration or an instance variable declaration of the form:

```
private JFileChooser chooser =  new JFileChooser(... starting-directory ...);
```

The "starting-directory" determines the folder on your disk whose contents will appear in the dialog box when it is first displayed. The user can navigate to other folders using the controls in the dialog, but it is convenient to start somewhere reasonable. If no starting directory is specified, the dialogue will start in the user's home directory. Using the expression

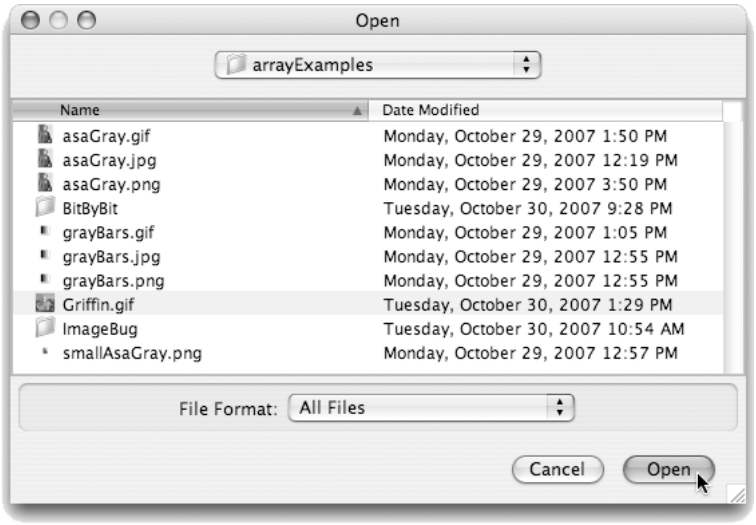

Figure 11.7: The `JFileChooser` "Open File" dialog box

```
new File( System.getProperty( "user.dir" ) )
```

will cause the dialog to start in the directory containing the program being executed.

The `File` class is a part of the standard java libraries used to represent file path names. The construction shown here creates a new `File` object from the `String` returned by asking the `System.getProperty` method to look up `"user.dir"`, a request to find the users' home directory. To use the `File` class, you will have to include an `import` for `"java.io.*"` in your .java file.

To display the dialog associated with a `JFileChooser`, a program must execute an invocation of the form

```
chooser.showOpenDialog( this )
```

Invoking `showOpenDialog` displays the dialog box and suspends the program's execution until the user clicks "Open" or "Cancel". The invocation returns a value indicating whether the user clicked "Open" or "Cancel". The value returned if "Open" is clicked is associated with the name `JFileChooser.APPROVE_OPTION`. Therefore, programs that use this method typically include the invocation in an `if` statement of the form

```
if ( chooser.showOpenDialog( this ) == JFileChooser.APPROVE_OPTION ) {
     ...  code to access and process the selected file ...
}
```

Finally, a `String` containing the name of the file selected by the user can be extracted from the `JFileChooser` using an invocation of the form

```
chooser.getSelectedFile().getAbsolutePath()
```

The `getSelectedFile` method asks the `JFileChooser` to return a `File` object describing the file chosen by the user. The `getAbsolutePath` method asks this `File` object to produce a `String` encoding the file's path name.

A program that uses a `JFileChooser` and an `SImage` constructor to load and display an image is shown in Figure 11.8. A picture of the interface produced by this program is shown in Figure 11.9.

```
import javax.swing.*;
import squint.*;
import java.awt.*;
import java.io.*;

// An image viewer that allows a user to select an image file and
// then displays the contents of the file on the screen
public class ImageAndLikeness extends GUIManager {
    // The dimensions of the program's window
    private final int WINDOW_WIDTH = 400, WINDOW_HEIGHT = 500;

    // Component used to display images
    private JLabel imageDisplay = new JLabel(  );

    // Dialog box through which user can select an image file
    private JFileChooser chooser =
          new JFileChooser( new File( System.getProperty( "user.dir" ) ));

    // Place the image display label and a button in the window
    public ImageAndLikeness() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

        contentPane.add( imageDisplay );
        contentPane.add( new JButton( "Choose an image file" ) );
    }

    // When the button is clicked, display image selected by user
    public void buttonClicked() {
        if ( chooser.showOpenDialog( this ) == JFileChooser.APPROVE_OPTION ) {

            String imageFileName = chooser.getSelectedFile().getAbsolutePath();

            SImage pic = new SImage( imageFileName );
            imageDisplay.setIcon( pic );
        }
    }
}
```

Figure 11.8: A simple image viewer

Figure 11.9: A picture of the Maguires displayed by the program in Figure 11.8

11.7 Think Negative

We have seen that we can use a constructor to turn an array of brightness values into an `SImage`. It is also possible to turn an `SImage` into an array containing the brightness values of its pixels using a method named `getPixelArray`. This method is the final tool we need to write programs that process images. We can now create an `SImage` from a file on our disk, get an array containing its brightness values, change the values in this array to brighten, rotate, scale, crop, or otherwise modify the image, and then create a new `SImage` from the modified array of values.

Perhaps the simplest type of image manipulation we can perform using these tools is a transformation in which each pixel's brightness value is replaced by a new value that is a function, f, of the original value. That is, for each position in our array, we set

```
pixels[x][y] = f( pixels[x][y] )
```

As an example of such a transformation, we will show how to convert an image into its own negative.

Long, long ago, before there were computers, MP3 players, and digital cameras, people took pictures using primitive cameras and light sensitive film like the examples shown in Figure 11.10. In fact, some photographers still use such strange devices.

The film used in non-digital cameras contains chemicals that react to light in such a way that, after being "developed" with other chemicals, the parts of the film that were not exposed to light become transparent while the areas that had been exposed to light remain opaque. As a result, after the film is developed, the image that is seen is bright where the actual scene was dark and dark where the scene was bright. These images are called *negatives*. Figure 11.11 shows an image of what the negative of the picture in Figure 11.1 might look like. As an example of image manipulation, we will write a program to modify the brightness values of an image's pixel so that the resulting values describe the negative of the original image. A sample of the interface our program will provide is shown in Figure 11.12.

Figure 11.10: Some antiques: A film camera and rolls of 120 and 35mm film

Figure 11.11: A negative image

Figure 11.12: Interface for a program that creates negative images

The function that describes how pixel values should be changed to produce a negative is simple. The value 0 should become 255, the value 255 should become 0, and everything in between should be scaled linearly. The appropriate function is therefore

$$f(x) = 255 - x$$

It is easy to apply this function to any single pixel. The statement

```
pixels[x][y] = 255 - pixels[x][y];
```

will do the job. All we need to do is use loops to execute this statement for every pair of possible `x` and `y` values.

To execute a statement for every possible value of `x` and `y`, we need some way to easily determine the correct range of index values for an image's table of pixels. The `SImage` class provides two methods that help. The methods `getWidth` and `getHeight` will return the number of columns and rows of pixels in an image repectively. Thus, we can use a loop of the form:

```
int x = 0;
while ( x < pixels.getWidth() ) {
    // Do something to all the pixels in column x
            ...
    x++;
}
```

to execute some statements for each possible `x` value. To execute some code for every possible combination of `x` and `y` values, we simply put a similar loop that steps through the `y` values inside of this loop. The result will look like:

```
int x = 0;
while ( x < pixels.getWidth() ) {

    int y = 0;
    while ( y < pixels.getHeight() ) {
        // Do something to pixels[x][y]
                              ...
        y++;
    }

    x++;
}
```

This is an example of a type of nested loop that is frequently used when processing two dimensional arrays.

The complete program to display negative images is shown in Figure 11.13. The nested loops are placed in the `buttonClicked` method between an instruction that uses `getPixelArray` to access the brightness values of the original image and a statement that uses an `SImage` construction to create a new image from the modified array of pixel values.

```
// A program that can display an image and its negative in a window
public class NegativeImpact extends GUIManager {
    private final int WINDOW_WIDTH = 450, WINDOW_HEIGHT = 360;

    // The largest brightness value used for a pixel
    private final int BRIGHTEST_PIXEL = 255;

    // Used to display the original image and the modified version
    private JLabel original = new JLabel(  ), modified = new JLabel(  );

    // Dialog box through which user can select an image file
    private JFileChooser chooser =  new JFileChooser(  );

    // Place two empty labels and a button in the window initially
    public NegativeImpact() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
        contentPane.add( original );
        contentPane.add( modified );
        contentPane.add( new JButton( "Show Images" ) );
    }

    // Let the user pick an image, then display the image and its negative
    public void buttonClicked() {
        if ( chooser.showOpenDialog( this ) == JFileChooser.APPROVE_OPTION ) {
            String imageFileName = chooser.getSelectedFile().getAbsolutePath();
            SImage pic = new SImage( imageFileName );
            original.setIcon( pic );

            // Replace every pixel's value by its negative equivalent
            int [][] pixels = pic.getPixelArray();
            int x = 0;
            while ( x < pic.getWidth() ) {
                int y = 0;
                while ( y < pic.getHeight() ) {
                    pixels[x][y] = 255 - pixels[x][y];
                    y++;
                }
                x++;
            }

            modified.setIcon( new SImage( pixels ) );
        }
    }
}
```

Figure 11.13: A program to display images and their negatives

11.8 `for` by `for`

Each of the nested loops used in Figure 11.13 has the general form:

```
int variable =  initial value;
while ( termination condition ) {
    ...  statement(s) to do some interesting work  ...
                        ⋮
    statement to change the value of variable;
}
```

Loops following this pattern are so common, that Java provides a shorthand notation for writing them. In Java, a statement of the form

```
for ( α; β; γ ) {
      σ
}
```

where α and γ are statements or local variable declarations,[4] β is any `boolean` expression, and σ is any sequence of 0 or more statements and declarations, is defined to be equivalent to the statements

```
α;
while ( β ) {
    σ;
    γ
}
```

A statement using this abbreviated form is called a `for` loop. In particular, the `for` loop

```
for ( int x = 0; x < pixels.getWidth(); x++ ) {
    ...   statement(s) to process column x
}
```

is equivalent to the `while` loop

```
int x = 0;
whilte ( x < pixels.getWidth() ) {
    ...   statement(s) to process column x
    x++;
}
```

To illustrate this, a version of the `buttonClicked` method from Figure 11.13 that has been revised to use `for` loops in place of its nested `while` loops is shown in Figure 11.14

We will be writing loops like this frequently enough that the bit of typing saved by using the `for` loop will be appreciated. More importantly, as you become familiar with this notation, the `for` loop has the advantage of placing three key components of the loop right at the beginning where they are easy to identify. These include:

[4] In all of our examples, α and γ will each be a single declaration or statement, but in general Java allows one to use lists of statements and declarations separated by commas where we have written α and γ.

```
// Let the user pick an image, then display the image and its negative
public void buttonClicked() {
    if ( chooser.showOpenDialog( this ) == JFileChooser.APPROVE_OPTION ) {
        String imageFileName = chooser.getSelectedFile().getAbsolutePath();
        SImage pic = new SImage( imageFileName );
        original.setIcon( pic );

        // Replace every pixel's value by its negative equivalent
        int [][] pixels = pic.getPixelArray();
        for ( int x = 0; x < pic.getWidth(); x++ ) {
            for ( int y = 0; y < pic.getHeight(); y++ ) {
                pixels[x][y] = 255 - pixels[x][y];
            }
        }

        modified.setIcon( new SImage( pixels ) );
    }
}
```

Figure 11.14: Code from Figure 11.13 revised to use `for` loops

- the initial value,
- the termination condition, and
- the way the loop moves from one step to the next

There is, by the way, no requirement that all the components of a `for` loop's header fit on one line. If the components of the header become complicated, is it good style to format the header so that they appear on separate lines as in:

```
for ( int x = 0;
      x < pixels.getWidth();
      x++ ) {
    ...   statement(s) to process column x
}
```

11.9 Moving Experiences

An alternative to changing the brightness values of an image's pixels is to simply move the pixels around. For example, most image processing programs provide the ability to rotate images or to flip an image vertically or horizontally. We can perform these operations by moving values from one position in a pixel array to another.

To start, consider how to write a program that can tip an image on its side by rotating the picture 90° counterclockwise. We will structure the program and its interface very much like the

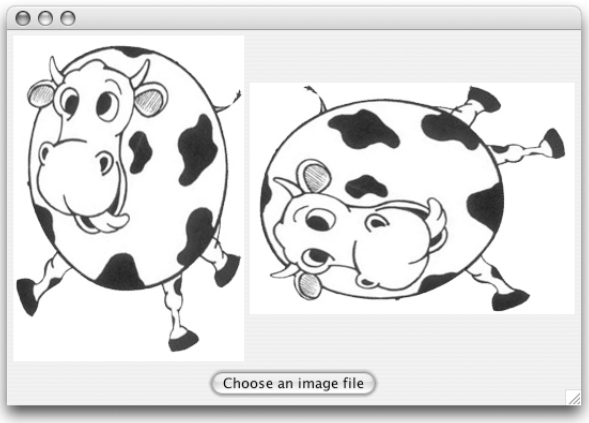

Figure 11.15: Window of a program that can rotate an image counterclockwise

program to display an image and its negative shown in Figure 11.13. When the user clicks the button in this new program's window, it will allow the user to select an image file, then it will display the original image and a rotated version of the image side by side in its window. A sample of this interface is shown if Figure 11.15. The only differences between the program that displayed negative images and this program will be found in the code that manipulates pixel values in the `buttonClicked` method.

If the original image loaded by this program is m pixels wide and n pixels high, then the rotated image will be n pixels wide and m pixels high. As a result, we cannot create the new image by just moving pixel values around in the array containing the original image's brightness values. We have to create a new array that has n columns and m rows. Assuming the original image is named `pic`, the needed array can be constructed in a local variable declaration:

```
int [][] result = new int[pic.getHeight()][pic.getWidth()];
```

Then, we can use a pair of nested loops to copy every value found in `pixels` into a new position within `result`.

At first glance, it might seem that we can move the pixels as desired by repeatedly executing the instruction

```
result[y][x] = pixels[x][y];          // WARNING:  This is not correct!
```

Unfortunately, if we use this statement, the image that results when our program is applied to the cow picture in Figure 11.15 will look like what we see in Figure 11.16. To understand why this would happen and how to rotate an image correctly, we need to do a little bit of analytic cow geometry.

Figure 11.17 shows an image of a cow and a correctly rotated version of the same image. Both are accompanied by x and y axes corresponding to the scheme used to number pixel values in an array describing the image. Let's chase the cow's tail as its pixels move from the original image on the left to the rotated image on the right.

In the original image, the y coordinates of the pixels that make up the tail fall between 20 and 30. If you look at the image on the right, it is clear that the x coordintes of the tail's new

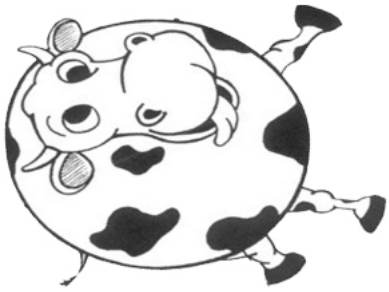

Figure 11.16: A cow after completing a double flip

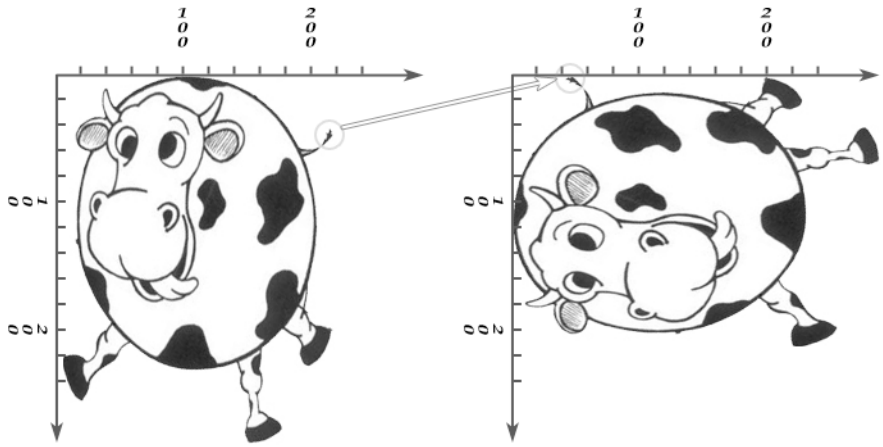

Figure 11.17: Geometry of pixel coordinate changes while rotating an image

position fall in the same range, 20-30. It seems as if y-coordinates from the original image become x-coordinates in the rotated image.

On the other hand, in the original image, the x coordinates of the pixels that make up the tail fall between 200 and 210. The y coordinates of the same pixels in the rotated version fall in a very different range, 0 to 10! Similarly, if you look at the x-coordinate of the edge of the cow's right ear in the original image it is about 10. In the rotated image, the edge of the same ear has an x coordinate of 200. It appears that small x coordinates become large y coordinates and large x coordinates become small y coordinates.

What is happening to the x coordinates is similar to what happened to brightness values when we made negative images. To convert the brightness value of a pixel into its negative value, we subtracted the original value from the maximum possible brightness value, 255. Similarly, to convert an x coordinate to a y coordinate in the rotated image, we need to subtract the x coordinate from the maximum possible x coordinate in the original image. The maximum x coordinate in our cow image is 210. Using this value, the x coordinate of the tip of the tail, 207, become $210 - 207 = 3$, a very small value as expected. Similarly, the x coordinate of the ear, 10, becomes $210 - 10 = 200$.

Therefore, the statement we should use to move pixel values to their new locations is

```
result[y][(pic.getWidth() - 1) - x] = pixels[x][y];
```

The expression used to describe the maximum x coordinte is

```
(pic.getWidth() - 1)
```

rather than `pic.getWidth()` since indices start at 0 rather than 1.

The key details for a program that uses this approach to rotate images are shown in Figure 11.18. As noted in the figure, this program is very similar to the code shown for generating a negative version of an image that we showed in Figure 11.13. In this example, however, we have placed the image manipulation code in a separate, `private` method. This is a matter of good programming style. It separates the details of image processing from the GUI interface, making both the new `rotate` method and the `buttonClicked` methods short and very simple.

Our explanation of why we use the expression

```
(picture.getWidth() - 1) - x
```

in the statement

```
result[y][(picture.getWidth() - 1) - x] = pixels[x][y];
```

suggests that it is possible to use a similar technique if we want to simply flip the pixels of an image horizontally or vertically. In fact, only two changes are required to convert the program shown in Figure 11.18 into a program that will flip an image horizontally. We would replace the statement in the body of the nested loops:

```
result[y][(picture.getWidth() - 1) - x] = pixels[x][y];
```

with:

```
result[(picture.getWidth() - 1) - x][y] = pixels[x][y];
```

In addition, the `result` array would now have to have exactly the same dimensions as the `pixels` array. Therefore, we would replace the declaration of `result` with

```
int [][] result = new int[picture.getWidth()][picture.getHeight()];
```

Finally, although not required, it would be good form to change the name of the method from `rotate` to `horizontalFlip`. The code for the new `horizontalFlip` method is shown in Figure 11.19. A sample of what the resulting program's display might look like is shown in Figure 11.20.

In the `rotate` method, it is clear that the `result` matrix needs to be separate from the `pixels` matrix because they have different dimensions. In the `horizontalFlip` method, the two arrays have the same dimensions. It is no longer clear that we need a separate `result` array. In the program to produce negative images, we made all of our changes directly in the `pixels` array. It might be possible to use the same approach in `horizontalFlip`. To try this we would remove the declaration of the `result` array and replace all references to `result` with the name `pixels`.

The revised `horizontalFlip` method is shown in Figure 11.21. Unfortunately, the program will not behave as desired. Instead, a sample of how it will modify the image selected by its user is shown in Figure 11.22.

To understand why this program does not behave as we might have hoped, think about what happens each time the outer loop is executed. The first time this loop is executed, `x` will be 0, so `(picture.getWidth() - 1) - x` will describe the index of the rightmost column of pixels.

```
// An program that allows a user to select an image file and displays
// the original and a verion rotated 90 degrees counterclockwise
public class BigTipper extends GUIManager {
    private final int WINDOW_WIDTH = 450, WINDOW_HEIGHT = 360;

            ⋮

    // Variable declarations and the constructor have been omitted to save space
    // They would be nearly identical to the declarations found in Figure 11.13

            ⋮

    // Rotate an image 90 degrees counterclockwise
    private SImage rotate( SImage picture ) {
        int [][] pixels = picture.getPixelArray();
        int [][] result = new int[picture.getHeight()][picture.getWidth()];

        for ( int x = 0; x < picture.getWidth(); x++ ) {
            for ( int y = 0; y < picture.getHeight(); y++ ) {
                result[y][(picture.getWidth() - 1) - x] = pixels[x][y];
           }
        }

        return new SImage( result );
    }

    // Display image selected by user and a copy that is rotated 90 degrees
    public void buttonClicked() {
        if ( chooser.showOpenDialog( this ) == JFileChooser.APPROVE_OPTION ) {

            String imageFileName = chooser.getSelectedFile().getAbsolutePath();

            SImage pic = new SImage( imageFileName );
            original.setIcon( pic );
            modified.setIcon( rotate( pic ) );
        }
    }
}
```

Figure 11.18: The `buttonClicked` method for a program to rotate images

```
// Flip an image horizontally
private SImage horizontalFlip( SImage picture ) {
    int [][] pixels = picture.getPixelArray();
    int [][] result = new int[picture.getWidth()][picture.getHeight()];

    for ( int x = 0; x < picture.getWidth(); x++ ) {
        for ( int y = 0; y < picture.getHeight(); y++ ) {
            result[(picture.getWidth() - 1) - x][y] = pixels[x][y];
       }
    }

    return new SImage( result );
}
```

Figure 11.19: A method to flip an image horizontally

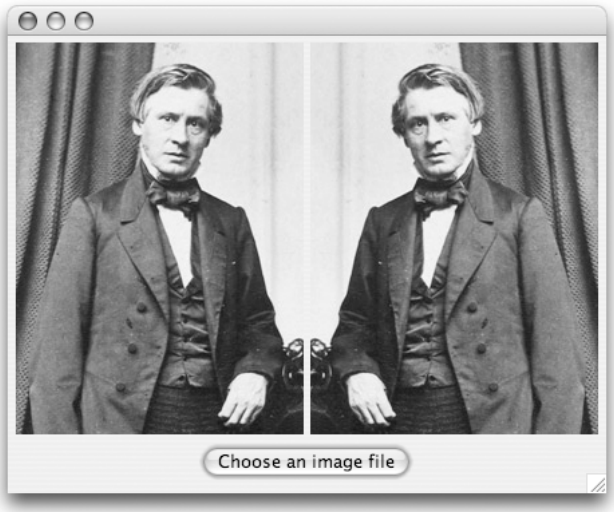

Figure 11.20: Asa Gray meets Asa Gray

```
// WARNING:  THIS METHOD IS INCORRECT!
private SImage horizontalFlip( SImage picture ) {
    int [][] pixels = picture.getPixelArray();

    for ( int x = 0; x < picture.getWidth(); x++ ) {
        for ( int y = 0; y < picture.getHeight(); y++ ) {
            pixels[(picture.getWidth() - 1) - x][y] = pixels[x][y];
       }
    }

    return new SImage( pixels );
}
```

Figure 11.21: A failed attempt to flip an image horizontally without a separate `result` array

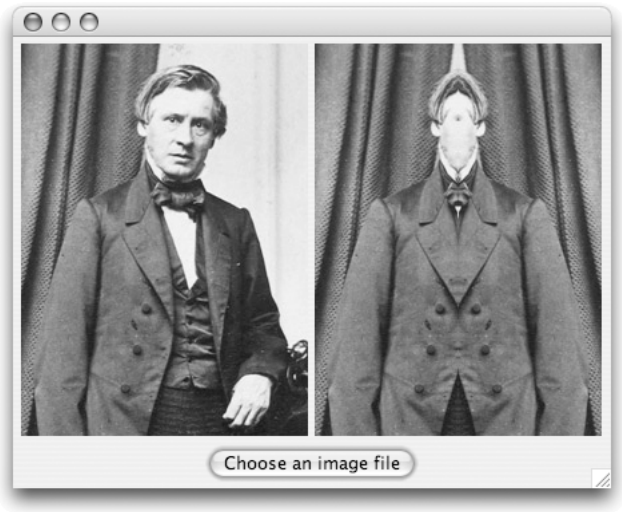

Figure 11.22: A Siamese twin?

Executing the inner loop will therefore copy all of the entries in the leftmost column of the image to the rightmost column. Next, when `x` is 1, the second column from the left will be copied to the second column from the right. As we progress to larger `x` values, columns from the right will appear on the left in reverse order. This sounds like exactly what we wanted.

If we started with the image shown in Figure 11.1, however, by the time the value of `x` reaches the mid point of the image, the numbers in the `pixels` array would describe the picture shown on the right in Figure 11.22. The left side of the original image has been correctly flipped and copied to the right side. In the process, however, the original contents of the right half of the image have been lost. Therefore, as the loop continues and copies columns from the right half of the image to the left, it will be copying columns it had earlier copied from the left half of the image rather than copying columns from the right half of the original image. In fact, it will copy these copies right back to where they came from! As a result, the remaining iterations of the loop will not appear to change anything. When the loop is complete, the image will look the same as it did when only half of the iterations had been executed.

This is a common problem when one writes code to interchange values within a single array. In many cases, the only solution is to use a second array like `result` to preserve all of the original values in the array while they are reorganized. In this case, however, is is possible to move the pixels as desired without an additional array. We just need an `int` variable to hold one original value while it is being interchanged with another.

To see how this is done, consider how the pixel values in the upper left and upper right corners of an image should be interchanged during the process of flipping an image horizontally. The upper left corner should move to the upper right corner, and the upper right should move to the upper left. If we try to do this using a pair of assignments like

```
pixels[picture.getWidth() - 1][0] = pixels[0][0];
pixels[0][0] = pixels[picture.getWidth() - 1][0];
```

we will end up with two copies of the value originally found in `pixels[0][0]` because the first assignment replaces the only copy of the original value of `pixels[picture.getWidth() - 1][0]` before it can be moved to the upper left corner. On the other hand, if we use an additional variable to save this value as in

```
int savePixel = pixels[picture.getWidth() - 1][0];
pixels[picture.getWidth() - 1][0] = pixels[0][0];
pixels[0][0] = savePixel;
```

the interchange will work correctly. If we perform such an interchange for every pair of pixels, the entire image can be flipped without using an additional array.

The code for a correct `horizontalFlip` method based on this idea is shown in Figure 11.23. Note that the termination condition for the outer for loop in this program is

```
x < picture.getWidth()/2
```

rather than

```
x < picture.getWidth()
```

Since each execution of the body of the loop interchanges two columns, we only need to execute the inner loop half as many times as the total number of columns. Can you predict what the program would do if we did not divide the width by 2 in this termination condition?

```
// Flip an image horizontally
private SImage horizontalFlip( SImage picture ) {
    int [][] pixels = picture.getPixelArray();

    for ( int x = 0; x < picture.getWidth()/2; x++ ) {
        for ( int y = 0; y < picture.getHeight(); y++ ) {
            int savePixel = pixels[(picture.getWidth() - 1) - x][y];
            pixels[(picture.getWidth() - 1) - x][y] = pixels[x][y];
            pixels[x][y] = savePixel;
        }
    }

    return( new SImage( pixels ) );
}
```

Figure 11.23: A method to flip a pixel array horizontally without creating a secondary array

11.10 Arrays of Arrays

Despite the fact that we have been showing programs that use two dimensional arrays, the Java language does not really include two dimensional arrays. The trick here is that technically Java only provides one dimensional arrays, but it is possible to define a one dimensional array of any type. We can have a one dimensional array of `JButton`s, a one dimensional array of `int`s, or even a one dimensional array of one dimensional arrays of `int`s. This is how we can write programs that appear to use two dimensional arrays. In Java's view, a two dimensional array is just a one dimensional array of one dimensional arrays.

A picture does a better job of explaining this than words. In Figure 11.4 we showed how the pixel values that describe the right eye from the photograph from Figure 11.1 could be visualized as a table. In most of the examples in this chapter, we have assumed that such a table was associated with a "two dimensional" array variable declared as

```
int [][]  pixels;
```

Figure 11.24 shows how the values describing the eye would actually be organized when stored in the `pixels` array. The array `pixels` shown in this figure is a one dimensional array containing 19 elements. Its elements are not `int`s. Instead, each of its elements is itself a one dimensional array containing 10 `int`s.

This is not just a way we can think about tables in Java, it is *the* way tables are represented using Java arrays. As a result, in addition to being able to access the `int`s that describe individual pixels in such an array, we can access the arrays that represent entire columns of pixels. For example, a statement like

```
int somePixel = pixels[x][y];
```

can be broken up into the two statements

```
int [] selectedColumn = pixels[x];
int somePixel = selectedColumn[y];
```

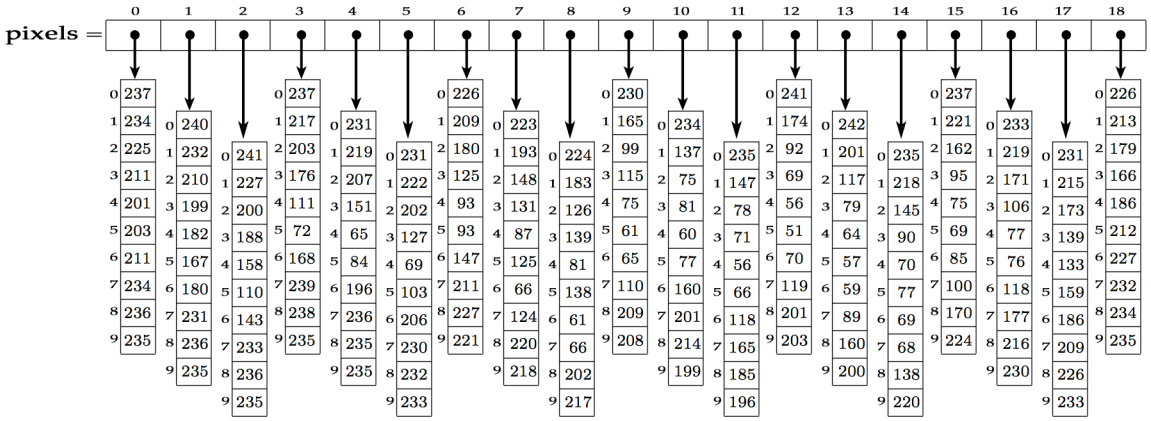

Figure 11.24: A two dimensional structure represented as an array of arrays

The expression `pixels[x]` extracts a single element from the array `pixels`. This element is the array that describes the entire xth column. The second line then extracts the yth element from the array that was extracted from `pixels`.

This fact about Java arrays has several consequences. We will discuss one that is quite simple but practical and one that is more subtle and a bit esoteric.

First, Java provides a way for a program to determine the number of elements in any array. If `x` is the name of an array, then `x.length` describes the number of elements in the array.[5] We did not introduce this feature earlier, because it is difficult to understand how to use this feature with a two dimensional array before understanding that two dimensional arrays are really just arrays of arrays. Suppose you think of `pixels` as a table like the one shown in Figure 11.4. What value would you expect `pixels.length` to produce? You might reasonably answer 19, 10, or even 190. The problem is that tables don't have lengths. Tables have widths and heights. On the other hand, if you realize that `pixels` is the name of an array of arrays as shown in Figure 11.24, then it is clear that `pixels.length` should produce 19. In general, if `x` is a two dimensional array, then `x.length` describes the width of the table.

It should also be clear how to use `length` to determine the height of a table. The height of an array of arrays is the length of any of the arrays that hold the columns. Thus, for our `pixels` array,

```
pixels[0].length
```

will produce the height of the table, 10. In general, if `x` is the name of an array representing a table, `x[0].length` gives the height of the table. Of course, for the `pixels` array, we could also use

[5]Unfortunately, while the designers of Java used the name `length` for both the mechanism used to determine the number of letters in a `String` and the size of an array, they made the syntax just a bit different. When used with a `String`, the name `length` must be followed by a pair of parentheses as in `word.length()`. When used with arrays, no parentheses are allowed.

```
// Flip an image horizontally
private SImage horizontalFlip( SImage picture ) {
        int [][] pixels = picture.getPixelArray();

        for ( int x = 0; x < pixels.length/2; x++ ) {
            for ( int y = 0; y < pixels[0].length; y++ ) {
                int savePixel = pixels[(pixels.length - 1) - x][y];
                pixels[(pixels.length - 1) - x][y] = pixels[x][y];
                pixels[x][y] = savePixel;
            }
        }

        return( new SImage( pixels ) );
    }
}
```

Figure 11.25: Using `.length` to control a loop

```
pixels[1].length
```

or

```
pixels[15].length
```

to describe the height of the table. We could not, however, use

```
pixels[25].length
```

as the index value 25 is out of range for the `pixels` array. In general, since using the index value 0 is more likely to be in range than any other value, it is a convention to say

```
x[0].length
```

to determine the height of a table.

The termination conditions in loops that process arrays often use `.length`. For example, Figure 11.25 shows the code of the `horizontalFlip` method revised to use `.length` rather than the `SImage` methods `getWidth` and `getHeight`.

The second consequence of the fact that two dimensional arrays are really arrays of arrays is that a program can build a two dimensional array piece by piece rather than all at once. This makes it possible to build arrays of arrays that cannot be viewed as simple, rectangular tables.

We know that the construction in the declaration

```
int [][] boxy = new int [5][7];
```

will produce an array with the structure shown in Figure 11.26. In Java, a construction of the form

```
new SomeType[size][]
```

creates an array with `size` elements each of which can refer to another array whose elements belong to `SomeType` without actually creating any arrays of `SomeType`. As a result, the declaration

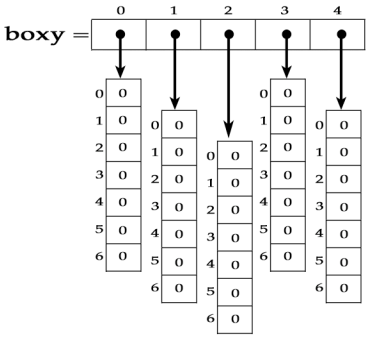

Figure 11.26: A small, rectangular array of arrays

```
int [][] boxy = new int[5][];
```

would create the five element array that runs across the top of Figure 11.26 but not create the five arrays containing zeroes that appear as the columns in that figure. The columns could then be created and associated with the elements of `boxy` by a loop of the form

```
for ( int x = 0; x < boxy.length; x++ ) {

    boxy[x] = new int[7];

}
```

Simple variations in the code of this loop can be used to produce two dimensional arrays that are not rectangular. For example, if we followed the declaration

```
int [][] trapezoid = new int[5][];
```

with a loop of the form

```
for ( int x = 0; x < trapezoid.length; x++ ) {

    trapezoid[x] = new int[ 3 + x ];

}
```

the structure created would look like the diagram in Figure 11.27. Each of the columns in this array is of a different length than the others. If we put the columns together to form a table we would end up with the collection shown in Figure 11.28.

There are other ways to associate element values with the entries of a two dimensional array that lead to even stranger structures. Consider the code

```
int [][] sharing = new int[5][];

for ( int x = 0; x < sharing.length - 1; x++ ) {
    sharing[x] = new int[ 7 ];
```

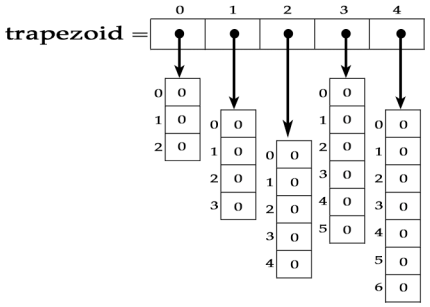

Figure 11.27: An array of arrays that is not rectangular

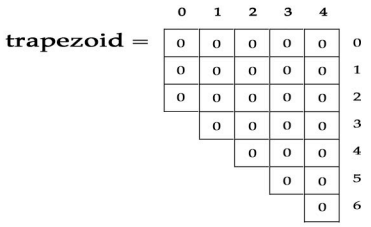

Figure 11.28: A non-rectangular table

```
}

sharing[ 4 ] = sharing[ 3 ];
```

This creates the structure shown in Figure 11.29. Here, two of the entries in the array of arrays are actually the same array.

In all of these diagrams we have shown the elements of `int` arrays filled with zeroes as they would be initialized by the computer. As a result, immediately after executing the code shown above, the elements `sharing[4][4]` and `sharing[3][4]` would both have the value 0. If we then executed the assignment

```
sharing[3][4] = 255;
```

the value associated with `sharing[3][4]` would be changed as expected. In addition, however, after this assignment, `sharing[4][4]`, would also have the value 255. Such behavior can make it very difficult to understand how a program works or why it doesn't. Therefore, we would discourage you from deliberately constructing such arrays. It is nevertheless important to understand that such structures are possible when debugging a program since they are sometimes created accidentally.

11.11 Summing Up

In the preceding sections we have examined examples of a variety of algorithms that process values in an array independently. There are many other algorithms that instead collect information about

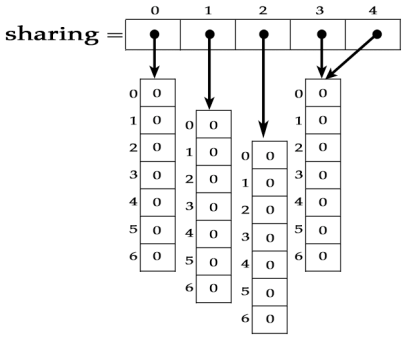

Figure 11.29: An array of arrays in which one element value is associated with two positions

all of the elements in an array or about some particular subgroup of elements. For example, when working with an array of pixel values we might want to find the brightest pixel value, the number of pixels brighter than 200, or the brightness value that appears most frequently in the top half of an image. As a simple example of such an algorithm, we will look at how to calculate the average brightness value of a specified square of pixels in an image.

This particular calculation has a nice application. Suppose we want to shrink an image by an integer factor. That is, we want to reduce the size of an image by 1/2, or 1/5, or in general 1/n. We can do this by computing the average brightness of non-overlapping, n by n squares of pixels in the original image and using the averages as the brightness values for pixels in an image whose width and height are 1/n th the width and height of the original.

Figure 11.30 illustrates the process we have in mind. At the top of the figure, we show a version of the familiar eye we have used in earlier examples. The image is enlarged so that individual pixels are visible. In addition, we have outlined 3 by 3 blocks of pixels. Because the image dimensions are not divisible by 3, pixels in the last row and column do not fall within any of these 3 by 3 blocks.

The bottom of the figure shows a version of the eye reduced to 1/3rd of its original size (but still enlarged so that individual pixels are visible). Each pixel in this smaller version corresponds to one of the 3 by 3 blocks of the original. The arrows in the figure connect the pixels in the leftmost 3 by 3 blocks of the original to the corresponding pixels in the reduced image. The brightness of each pixel in the reduced image was determined by averaging the values of the nine pixels in the corresponding block of the original image.

This is not the ideal way to reduce an image. As this example illustrates, it completely ignores the pixels that don't fall in any of the square blocks. It does, however, produce visually reasonable results. For example, Figure 11.31 show the results of applying this technique to produce reduced versions of another image we have seen before ranging from half size to 1/12.

A key step in implementing such a scaling algorithm is writing code to compute the average of a specified block of pixels. The block can be specified by giving the indices of its upper left corner and its size. To add up the values in such a block, we will use a pair of doubly nested loops to iterate over all pairs of x and y coordinates that fall within the block. These loops will look very much like the nested loops we have seen in early examples except that they will not start at 0 and they must stop when they reach the edges of the block rather than the edges of the entire pixel array. Once we have the sum of all pixel values, we can compute the average by simply dividing

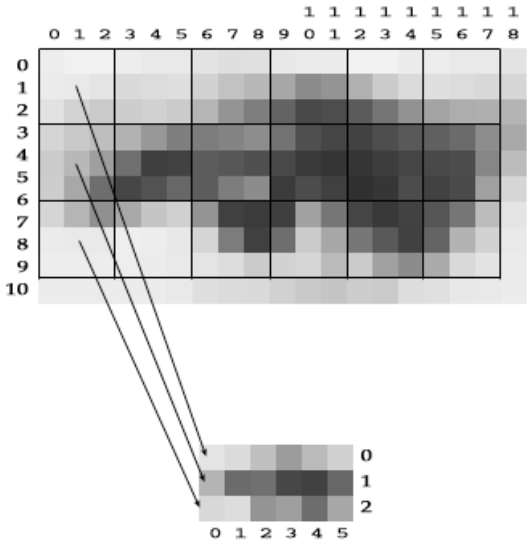

Figure 11.30: Scaling an image by averaging blocks of pixel brightnesses

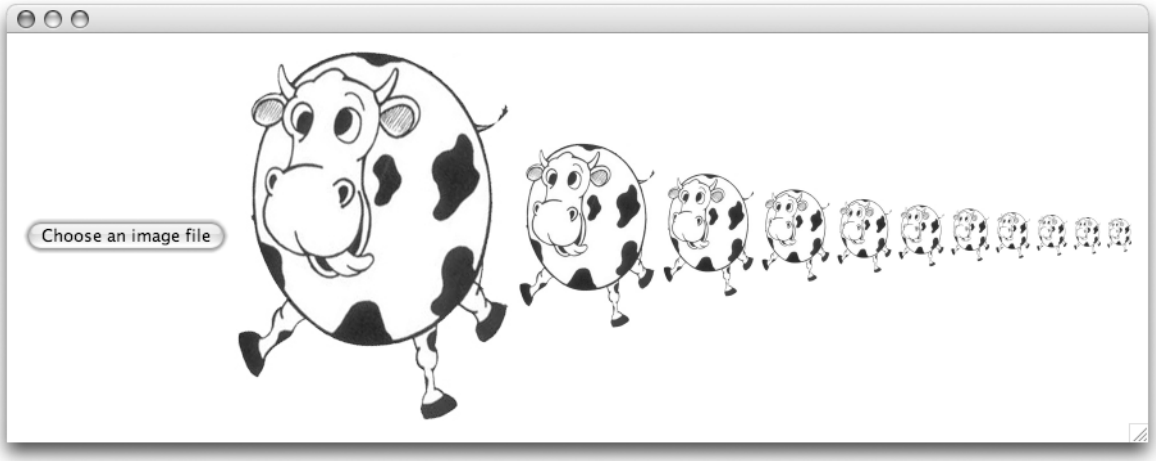

Figure 11.31: Scaling an image repeatedly

```
// Determine average brightness of a block of size by size pixels
// at position (left, top)
private int average( int [][] pixels, int left, int top, int size ) {
    int sum = 0;

    for ( int x = left; x < left + size; x++ ) {
        for ( int y = top; y < top + size; y++ ) {
            sum = sum + pixels[x][y];
        }
    }
    return sum/(size*size);
}
```

Figure 11.32: A method to calculate the average brightness of a block of pixels

```
// Create a copy of an image reduced to 1/s of its original size
private SImage scale( SImage original, int s ) {
    int [][] pixels = original.getPixelArray();
    int [][] result = new int[ pixels.length/s][pixels[0].length/s ];

    for ( int x = 0; x < result.length; x++ ) {
        for ( int y = 0; y < result[0].length; y++ ) {
            result[x][y] = average( pixels, s*x, s*y, s );
        }
    }
    return new SImage( result );
}
```

Figure 11.33: A method to scale an image to 1/s its original size

the sum by the number of pixels in the block.

The code for a method to compute block averages in this way is shown in Figure 11.32. The location of the block to be averaged is provided through the parameters `left` and `top` that specify the coordinates of the left and top edges of the block. The parameter `size` determines how wide and high the block should be. As a result, the expressions `left + size` and `top + size` describe the coordinates of the right and bottom edges of the block. These expressions are used in the termination conditions of the loops in the method body.

Given the `average` method, it is easy to write a method to scale an entire image. The code for such a method is shown in Figure 11.33. The second parameter, `s`, specifies the desired scaling factor. Therefore, the width and height of the resulting image can be determined by dividing the width and height of the original by `s`. The method begins by creating an array `result` using the reduced width and height. The body of the loop then fills the entries of this `result` array by repeatedly invoking `average` on the appropriate block of original pixel values.

The complete code of the program that produced the image shown in Figure 11.31 is shown in

Figure 11.34. It repeatedly applies the `scale` method for scaling factors ranging from 1 to 1/12th and displays each scaled image within a separate `JLabel` in its window.

11.12 Purple Cows?

Some of you may have already recognized that something is missing in the image shown in Figure 11.31. The cows are not purple! If not, consider the image of lego robots processed by our image scaling program shown in Figure 11.35. If you are reading a printed/copied version of this text, then the robots will all appear to be constructed out of gray Legos. It is possible to get gray Legos. In fact, a few of the pieces of the robot pictured in the figure actually are gray. Most of the pieces, however, are more typical Lego colors: bright reds, blues and greens.

Although we have focused on how to manipulate grayscale images, the `SImage` class does provide the ability to handle color images. In this section, we will show you how to write programs designed to handle color images.

First, when you load the contents of image file that describes a color picture using a construction like the one in the following variable declaration

```
SImage pic = new SImage( "colorfulRobot.jpg" );
```

the `SImage` you create retains the color information. If `imageLabel` is the name of a `JLabel` in your program's window and you display the `SImage` by executing

```
original.setIcon( pic );
```

it will appear in color on your screen.

The colors vanish when you access the information about individual pixels by invoking the `getPixelArray` method. This method returns an array of values that only describe the brightnesses of the pixels, not their colors. Fortunately, `SImage` provides several ways to get information about pixel colors.

As mentioned early in this chapter, one common scheme for describing the colors in a digital image is to describe how a color can be mixed by combining red, green and blue light. This is typically done by describing the brightness of each color component using a number between 0 and 255 much as we have been using such values to describe overall brightness. This is known as the RGB color scheme.

`SImage` provides three methods that can be used to access the brightness values for image pixels corresponding to each of these three primary colors. In particular, an invocation of the form

```
int [][] redBrightnesses = pic.getRedPixelArray();
```

will return an array of values describing the "redness" of the pixels in an image. Similar methods named `getGreenPixelArray` and `getBluePixelArray` can be used to access the "greenness" and "blueness" values.

The `SImage` class also provides a constructor designed for programs that want to modify the values describing the redness, greenness and blueness of an image's pixels. This constructor takes three tables of pixel values as parameters. All three tables must have the same dimensions. The first table is interpreted as the redness values for a new image's pixels. The second and third are interpreted as the greenness and blueness values respectively. Therefore, if we declare

```
// A program that allows a user to select an image file and displays
// the original and a series of copies ranging from 1/2 to 1/12 the original
public class CheaperByTheDozen extends GUIManager {
    // The dimensions of the program's window
    private final int WINDOW_WIDTH = 870, WINDOW_HEIGHT = 380;

    // Dialog box through which user can select an image file
    private JFileChooser chooser =
             new JFileChooser( new File( System.getProperty( "user.dir" ) ));

    // Place the image display labels and a button in the window
    public CheaperByTheDozen() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

        JButton but = new JButton( "Choose an image file" );
        contentPane.add( but );
    }

    // Determine average brightness of a block of size by size pixels
    // at position (left, top)
    private int average( int [][] pixels, int left, int top, int size ) {
        // See Figure 11.32 for method's body
    }

    // Create a copy of an image reduced to 1/s of its original size
    private SImage scale( SImage original, int s ) {
        // See Figure 11.33 for method's body
    }

    // Display image selected by user and copies of reduced sizes
    public void buttonClicked() {
       if ( chooser.showOpenDialog( this ) == JFileChooser.APPROVE_OPTION ) {
          SImage pic = new SImage( chooser.getSelectedFile().getAbsolutePath() );

          for ( int s = 1; s < 12; s++ ) {
              JLabel modified = new JLabel( );
              modified.setIcon(  scale( pic, s ) );
              contentPane.add( modified );
          }
       }
    }
}
```

Figure 11.34: A program to display a series of scaled copies of an image

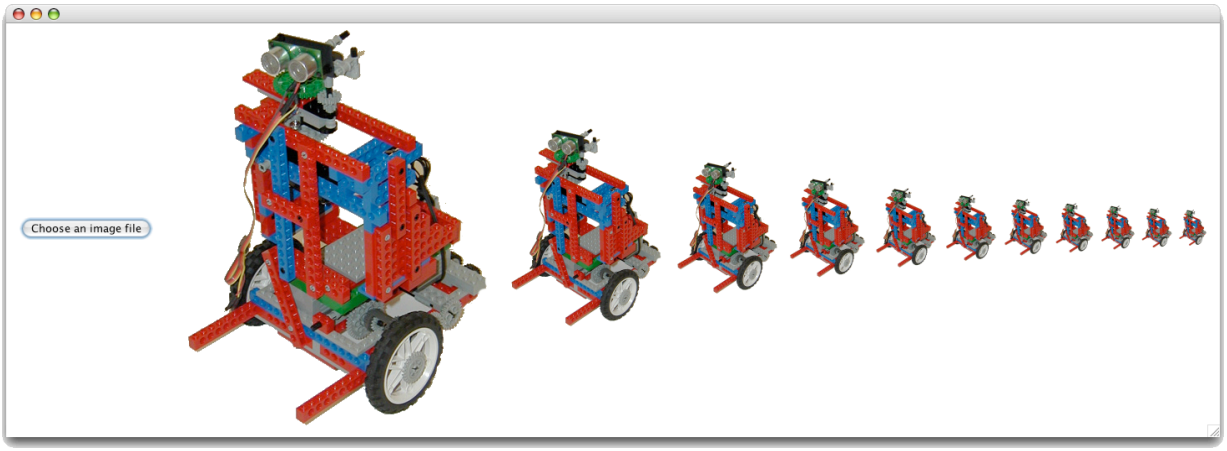

Figure 11.35: Red is grey and yellow white, but we decide which is right...

```
// Create a copy of a color image reduced to 1/s of its original size
private int [][] scaleBrighnesses( int [][] pixels, int s ) {
    int [][] result = new int[ pixels.length/s][pixels[0].length/s ];

    for ( int x = 0; x < result.length; x++ ) {
        for ( int y = 0; y < result[0].length; y++ ) {
            result[x][y] = average( pixels, s*x, s*y, s );
        }
    }
    return result;
}
```

Figure 11.36: A method to scale a table of pixel brightness values

```
int [][] rednesses, greennesses, bluenesses;
```

as the names of two dimensional arrays and associate these names with appropriate arrays of `int`s, the construction

```
new SImage( rednesses, greennesses, bluenesses )
```

can be used to create an image described by the contents of the arrays.

To illustrate how these mechanisms can be used, consider how we can revise our code for scaling images to work for color images instead of just grayscale images. The basic idea is to break the grayscale image scaling code found in Figure 11.33 into two parts. The first statement in the method is specific to processing the gray levels of an image. The `for` loops that form the remainder of the method correctly describe how to scale an array of brightness values whether they describe pixel graynesses, rednesses, greennesses, or bluenesses. Therefore, the first step is to move these for loops into a separate method designed to process an array of brightness values independent of its source. This method, named `scaleBrighnesses` is shown if Figure 11.36. Most of its code is

```
// Create a copy of an image reduced to 1/s of its original size
private SImage scale( SImage picture, int s ) {
    int [][] rednesses = picture.getRedPixelArray( );
    int [][] greennesses = picture.getGreenPixelArray( );
    int [][] bluenesses = picture.getBluePixelArray( );

    rednesses = scaleBrighnesses( rednesses, s );
    greennesses = scaleBrighnesses( greennesses, s );
    bluenesses = scaleBrighnesses( bluenesses, s );

    return new SImage( rednesses, greennesses, bluenesses );
}
```

Figure 11.37: A colorful image scaling method

identical to the `scale` method from Figure 11.33. The big changes are that it takes and returns two dimensional arrays of `int`s rather than `SImage`s. Note in particular, that this method continues to use the unmodified `average` method. As originally written, the `average` method did not depend on the fact that the array it was processing contained grayness values.

Given the `scaleBrighnesses` method, it is easy to write a `scale` method that handles colored images. We simply get the arrays that describe the image's redness, greenness, and blueness, process each of them using `scaleBrightnesses`, and finally construct a new image with the results. The code to do this is shown in Figure 11.37.

For those who want to replace the somewhat repetitive code in Figure 11.37 with some loops, the `SImage` class provides an alternate method of working with the three arrays describing the colors that compose an image. First, the SImage class provides names for the three colors that make up an image, `SImage.RED`, `SImage.GREEN`, and `SImage.BLUE`. In addition, `SImage` provides a method named `getPixelArrays` that returns a three dimensional array of `int`s. That is, it can be invoked in a statement of the form

```
int [][][] brightnesses = picture.getPixelArrays();
```

If `brightnesses` is declared in this way, then

```
brightnesses[SImage.RED]
```

will be the array that would have been obtained by invoking

```
picture.getRedPixelArray()
```

The expressions `brightnesses[SImage.GREEN]` and `brightnesses[SImage.BLUE]` would similarly produce two dimensional arrays describing the greenness and blueness of image pixels. Finally, there is an `SImage` constructor that accepts a three dimensional array like `brightnesses` as a parameter.

The names `SImage.RED`, `SImage.GREEN`, and `SImage.BLUE` actually refer to consecutive `int` values. As a result, they can be used to write loops to process the three pixel arrays that describe an image. Figure 11.38 shows an alternate version of the `scale` method that uses such a loop.

```
private SImage scale( SImage picture, int s ) {
    int [][][] pixels = picture.getPixelArrays();
    for ( int c = SImage.RED; c <= SImage.BLUE; c++ ) {
        pixels[c] = scaleBrighnesses( pixels[c], s );
    }
    return new SImage( pixels );
}
```

Figure 11.38: Scaling with a three dimensional array and a loop

11.13 Summary

In this chapter, we have introduced the basic mechanisms required to work with arrays in Java. While our focus has been on arrays that represent digital images, the techniques we have explored are applicable to arrays of any kind. We have seen how to declare array variables, construct arrays, and access individual elements of arrays. We have also provided examples to illustrate several of the most important patterns used when writing code to manipulate data within an array. We have seen how to apply a transformation independently to each element of an array, how to reorganize an array by rearranging its elements, and how to collect summary information about a subset of an array's elements. These techniques are fundamental when working with arrays of many types, not just with pixel arrays. We will return to examine more sophisticated array processing techniques in a later chapter.

In addition to the material on arrays, we introduced mechanisms for working with files in general and with image files in particular. We introduced the `JFileChooser` which makes it easy to provide a flexible interface through which a program's user can select a file for processing. We also saw how image files could be accessed using `SImage` constructors. Finally, we saw how `JLabel`s can be used to display images within a program's GUI interface.