UNIT III BASICS OF DATA VISUALIZATION

The Seven Stages of Visualizing Data - Getting Started with Processing - Mapping - Time Series - Connections and Correlations - Scatterplot Maps - Trees, Hierarchies, and Recursion - Networks and Graphs – Acquiring Data – Parsing Data.

THE SEVEN STAGES OF VISUALIZING DATA

The process of understanding data begins with a set of numbers and a question. The following steps form a path to the answer:

Acquire

Obtain the data, whether from a file on a disk or a source over a network.

Parse

Provide some structure for the data's meaning, and order it into categories.

Filter

Remove all but the data of interest.

Mine

Apply methods from statistics or data mining as a way to discern patterns or place the data in mathematical context.

Represent

Choose a basic visual model, such as a bar graph, list, or tree.

Refine

Improve the basic representation to make it clearer and more visually engaging.

Interact

Add methods for manipulating the data or controlling what features are visible

An Example

To illustrate the seven steps listed in the previous section, and how they contribute to effective information visualization, let's look at how the process can be applied to understanding a simple data set. In this case, we'll take the zip code numbering system that the U.S. Postal Service uses. The application is not particularly advanced, but it provides a skeleton for how the process works.

What Is the Question?

All data problems begin with a question and end with a narrative construct that provides a clear answer. The Zipdecode project was developed out of a personal interest in the relationship of the zip code numbering system to geographic areas. Living in Boston, I knew that numbers starting with a zero denoted places on the East Coast. Having spent time in San Francisco, I knew the initial numbers for the West Coast were all nines. I grew up in Michigan, where all our codes were four-prefixed. But what sort of area does the second digit specify? Or the third?

The finished application was initially constructed in a few hours as a quick way to take what might be considered a boring data set (a long list of zip codes, towns, and their latitudes and longitudes) and create something engaging for a web audience that explained how the codes related to their geography.

Acquire

The acquisition step involves obtaining the data. Like many of the other steps, this can be either extremely complicated (i.e., trying to gather useful data from a large system) or very simple (reading a readily available text file). A copy of the zip code listing can be found on the U.S. Census Bureau web site, as it is frequently used for geographic coding of statistical data. The listing is a freely available file with approximately 42,000 lines, one for each of the codes, a tiny portion of which is shown in Figure 1-1.

```
00210    +43.005895    -071.013202    U    PORTSMOUTH    33    015
00211    +43.005895    -071.013202    U    PORTSMOUTH    33    015
00212    +43.005895    -071.013202    U    PORTSMOUTH    33    015
00213    +43.005895    -071.013202    U    PORTSMOUTH    33    015
00214    +43.005895    -071.013202    U    PORTSMOUTH    33    015
00215    +43.005895    -071.013202    U    PORTSMOUTH    33    015
00501    +40.922326    -072.637078    U    HOLTSVILLE    36    103
00544    +40.922326    -072.637078    U    HOLTSVILLE    36    103
00601    +18.165273    -066.722583         ADJUNTAS      72    001
00602    +18.393103    -067.180953         AGUADA        72    003
00603    +18.455913    -067.145780         AGUADILLA     72    005
00604    +18.493520    -067.135883         AGUADILLA     72    005
00605    +18.465162    -067.141486    P    AGUADILLA     72    005
00606    +18.172947    -066.944111         MARICAO       72    093
00610    +18.288685    -067.139696         ANASCO        72    011
00611    +18.279531    -066.802170    P    ANGELES       72    141
00612    +18.450674    -066.698262         ARECIBO       72    013
00613    +18.458093    -066.732732    P    ARECIBO       72    013
00614    +18.429675    -066.674506    P    ARECIBO       72    013
00616    +18.444792    -066.640678         BAJADERO      72    013
```

Figure 1-1. Zip codes in the format provided by the U.S. Census Bureau

Acquisition concerns how the user downloads your data as well as how you obtained the data in the first place. If the final project will be distributed over the Internet, as you design the application, you have to take into account the time required to download data into the browser. And because data downloaded to the browser is probably part of an even larger data set stored on the server, you may have to structure the data on the server to facilitate retrieval of common subsets.

Parse

After you acquire the data, it needs to be parsed—changed into a format that tags each part of the data with its intended use. Each line of the file must be broken along its individual parts; in this case, it must be delimited at each tab character. Then, each piece of data needs to be converted to a useful format. Figure 1-2 shows the layout of each line in the census listing, which we have to understand to parse it and get out of it what we want.

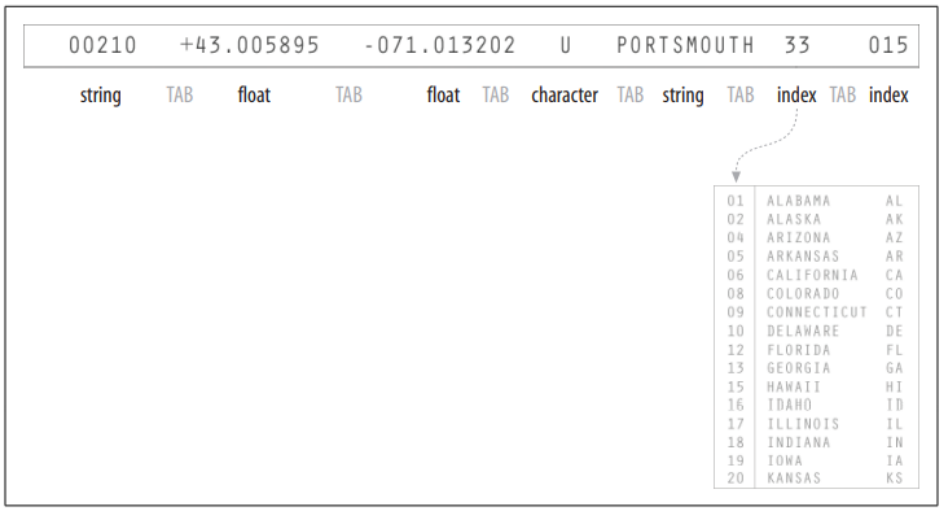


Figure 1-2. Structure of acquired data

Each field is formatted as a data type that we'll handle in a conversion program:

String

A set of characters that forms a word or a sentence. Here, the city or town name is designated as a string. Because the zip codes themselves are not so much numbers as a series of digits (if they were numbers, the code 02139 would be stored as 2139, which is not the same thing), they also might be considered strings.

Float

A number with decimal points (used for the latitudes and longitudes of each location). The name is short for floating point, from programming nomenclature that describes how the numbers are stored in the computer's memory.

Character

A single letter or other symbol. In this data set, a character sometimes designates special post offices.

Integer

A number without a fractional portion, and hence no decimal points (e.g., –14, 0, or 237).

Index

Data (commonly an integer or string) that maps to a location in another table of data. In this case, the index maps numbered codes to the names and two-digit abbreviations of states. This is common in databases, where such an index is used as a pointer into another table, sometimes as a way to compact the data further (e.g., a two-digit code requires less storage than the full name of the state or territory).

With the completion of this step, the data is successfully tagged and consequently more useful to a program that will manipulate or represent it in some way.

Filter

The next step involves filtering the data to remove portions not relevant to our use. In this example, for the sake of keeping it simple, we'll be focusing on the contiguous 48 states, so the records for cities and towns that are not part of those states— Alaska, Hawaii, and territories such as Puerto Rico—are removed. Another project could require significant mathematical work to place the data into a mathematical model or normalize it.

Mine

This step involves math, statistics, and data mining. The data in this case receives only a simple treatment: the program must figure out the minimum and maximum values for latitude and longitude by running through the data (as shown in Figure 1-3) so that it can be presented on a screen at a proper scale. Most of the time, this step will be far more complicated than a pair of simple math operations.

Represent

This step determines the basic form that a set of data will take. Some datasets are shown as lists, others are structured like trees, and so forth. In this case, each zip code has a latitude and longitude, so the codes can be mapped as a two-dimensional plot, with the minimum and maximum values for the latitude and longitude used for the start and end of the scale in each dimension. This is illustrated in Figure 1-4.

The Represent stage is a linchpin that informs the single most important decision in a visualization project and can make you rethink earlier stages. How you choose to represent the data can influence the very first step (what data you acquire) and the third step (what particular pieces you extract).

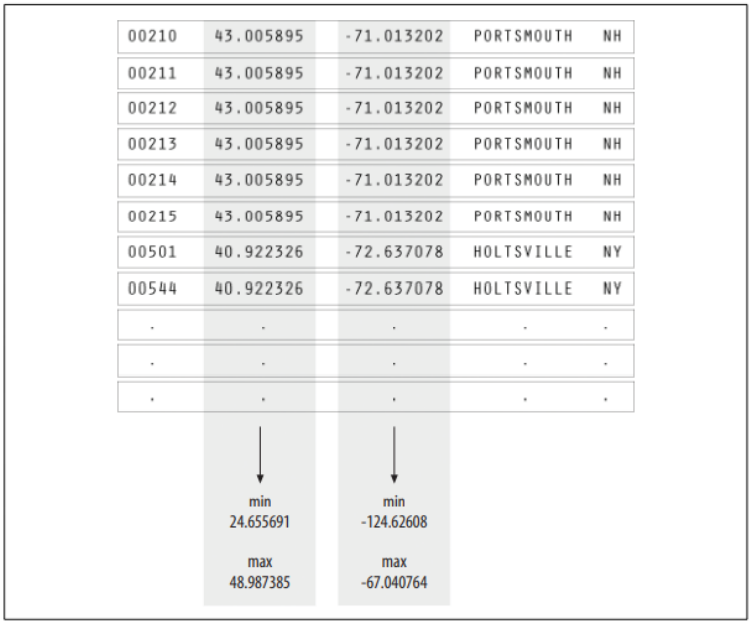


Figure 1-3. Mining the data: just compare values to find the minimum and maximum

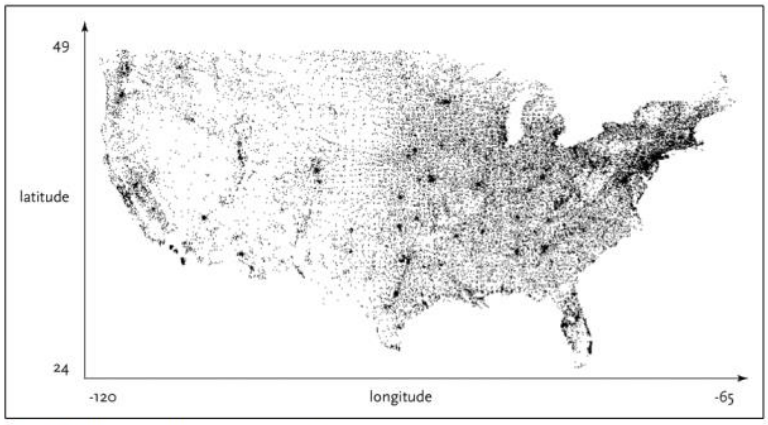


Figure 1-4. Basic visual representation of zip code data

Refine

In this step, graphic design methods are used to further clarify the representation by calling more attention to particular data (establishing hierarchy) or by changing attributes (such as color) that contribute to readability.

Hierarchy is established in Figure 1-5, for instance, by coloring the background deep gray and displaying the selected points (all codes beginning with four) in white and the deselected points in medium yellow.

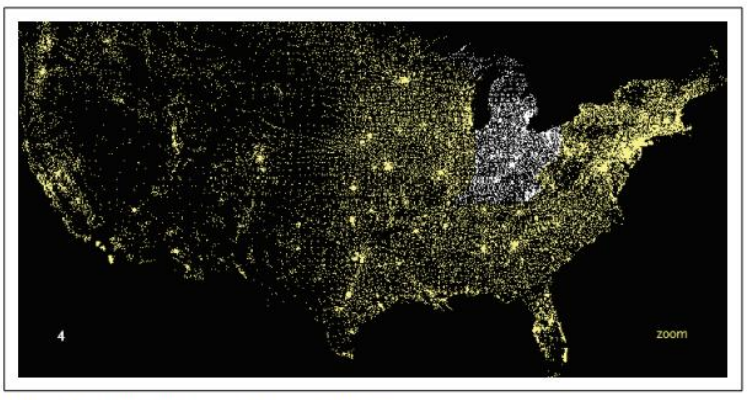


Figure 1-5. Using color to refine the representation

Interact

The next stage of the process adds interaction, letting the user control or explore the data. Interaction might cover things like selecting a subset of the data or changing the viewpoint. As another example of a

stage affecting an earlier part of the process, this stage can also affect the refinement step, as a change in viewpoint might require the data to be designed differently.

In the Zipdecode project, typing a number selects all zip codes that begin with that number. Figures 1-6 and 1-7 show all the zip codes beginning with zero and nine, respectively.

Another enhancement to user interaction enables the users to traverse the display laterally and run through several of the prefixes. After typing part or all of a zip code, holding down the Shift key allows users to replace the last number typed without having to hit the Delete key to back up.

Figure 1-6. The user can alter the display through choices (zip codes starting with 0)

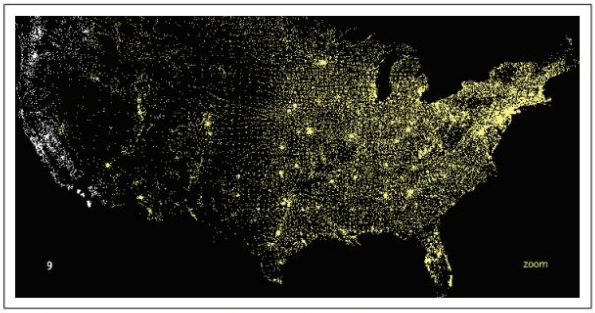


Figure 1-7. The user can alter the display through choices (zip codes starting with 9)

Iteration and Combination

Figure 1-12 shows the stages in order and demonstrates how later decisions commonly reflect on earlier stages. Each step of the process is inextricably linked because of how the steps affect one another. In the Zipdecode application, for instance:

• The need for a compact representation on the screen lead to refilter the data to include only the contiguous 48 states.

• The representation step affected acquisition because after developed the application, it was modified so it could show data that was downloaded over a slow Internet connection to the browser. The change to the structure of the data allows the points to appear slowly, as they are first read from the data file, employing the data itself as a "progress bar."

• Interaction by typing successive numbers meant that the colors had to be modified in the visual refinement step to show a slow transition as points in the display are added or removed. This helps the user maintain context by preventing the updates on-screen from being too jarring.

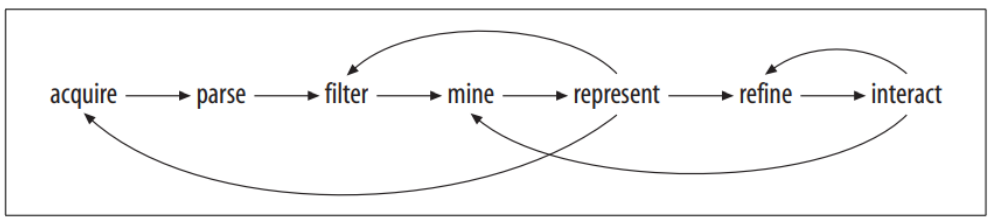


Figure 1-12. Interactions between the seven stages

The connections between the steps in the process illustrate the importance of the individual or team in addressing the project as a whole. This runs counter to the common fondness for assembly-line style

projects, where programmers handle the technical portions, such as acquiring and parsing data, and visual designers are left to choose colors and typefaces. At the intersection of these fields is a more interesting set of properties that demonstrates their strength in combination.

When acquiring data, consider how it can change, whether sporadically (such as once a month) or continuously. This expands the notion of graphic design that's traditionally focused on solving a specific problem for a specific data set, and instead considers the meta-problem of how to handle a certain kind of data that might be updated in the future.

In the filtering step, data can be filtered in real time, as in the Zipdecode application. During visual refinement, changes to the design can be applied across the entire system. For instance, a color change can be automatically applied to the thousands of elements that require it, rather having to make such a tedious modification by hand. This is the strength of a computational approach, where tedious processes are minimized through automation.

GETTING STARTED WITH PROCESSING

Processing is a simple programming environment that was created to make it easier to develop visually oriented applications with an emphasis on animation and provide users with instant feedback through interaction. As its capabilities have expanded over the past six years, Processing has come to be used for more advanced production-level work in addition to its sketching role.

Processing is based on Java, but because program elements in Processing are fairly simple, we can learn to use it from this book even if we don't know Java. If we're familiar with Java, it's best to forget that Processing has anything to do with it for a while, at least until we get the hang of how the API works. The latest version of Processing can be downloaded at:

http://processing.org/download

Processing consists of:

• The Processing Development Environment (PDE). This is the software that runs when you double-click the Processing icon. The PDE is an Integrated Development Environment with a minimalist set of features designed as a simple introduction to programming or for testing one-off ideas.

• A collection of commands (also referred to as functions or methods) that make up the "core" programming interface, or API, as well as several libraries that support more advanced features, such as drawing with OpenGL, reading XML files, and saving complex imagery in PDF format.

• A language syntax, identical to Java but with a few modifications.

• An active online community, hosted at http://processing.org

Sketching with Processing

A Processing program is called a sketch. The idea is to make Java-style programming feel more like scripting, and adopt the process of scripting to quickly write code. Sketches are stored in the sketchbook, a folder that's used as the default location for saving all of your projects. When you run Processing, the sketch last used will automatically open. If this is the first time Processing is used (or if the sketch is no longer available), a new sketch will open.

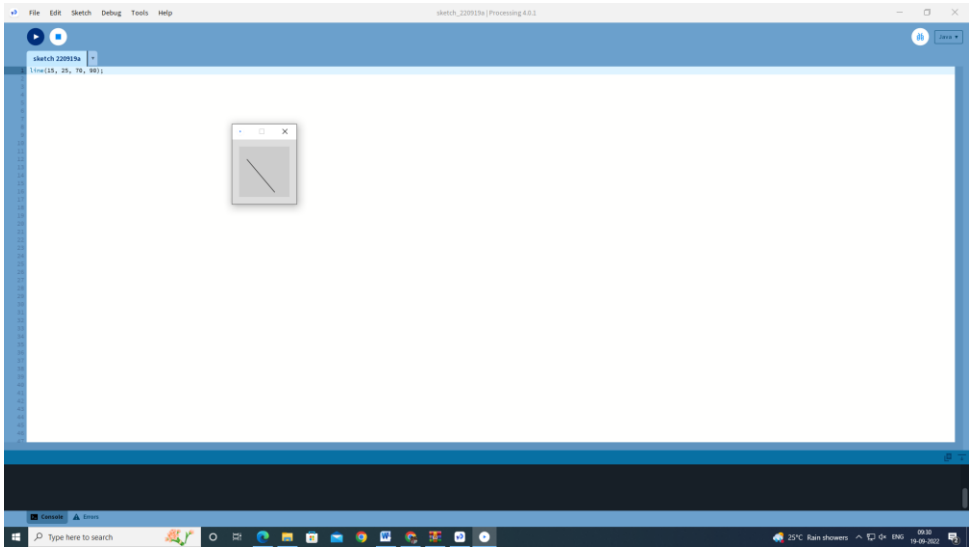
Sketches that are stored in the sketchbook can be accessed from File → Sketchbook. Alternatively, File → Open... can be used to open a sketch from elsewhere on the system.

Hello World

Programming languages are often introduced with a simple program that prints “Hello World” to the console. The Processing equivalent is simply to draw a line:

```
line(15, 25, 70, 90);
```

Enter this example and press the Run button, which is an icon that looks like the Play button on any audio or video device. The result will appear in a new window, with a gray background and a black line from coordinate (15, 25) to (70, 90).

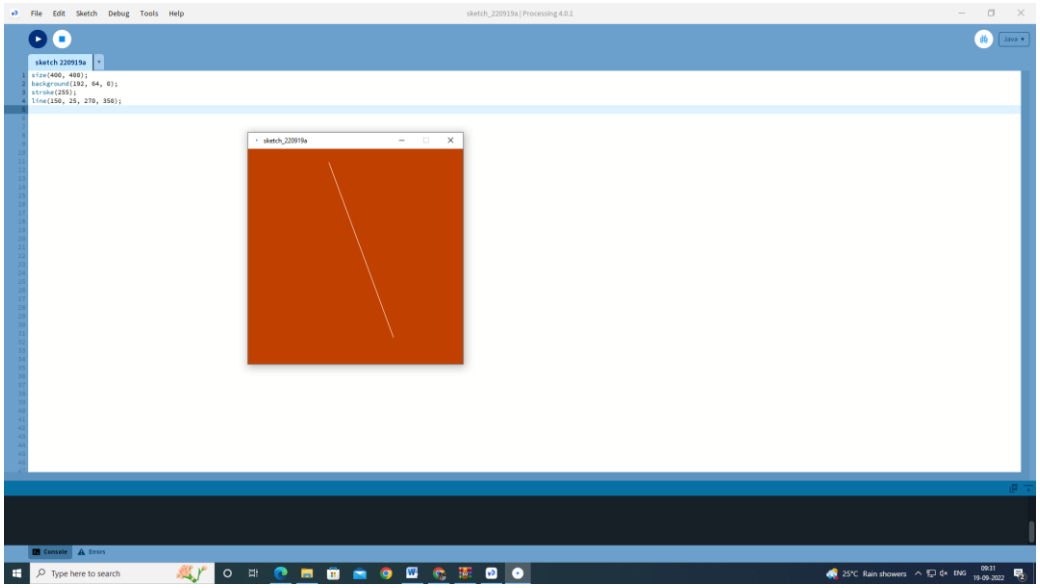


The (0, 0) coordinate is the upper-lefthand corner of the display window. Building on this program to change the size of the display window and set the background color, type in the code from Example 2-1.

Example 2-1. Simple sketch

```
size(400, 400);
background(192, 64, 0);
stroke(255);
line(150, 25, 270, 350);
```

This version sets the window size to 400 × 400 pixels, sets the background to an orange-red, and draws the line in white, by setting the stroke color to 255. By default, colors are specified in the range 0 to 255.



Other variations of the parameters to the stroke( ) function provide alternate results:

```
stroke(255); // sets the stroke color to white
```

```
stroke(255, 255, 255); // identical to stroke(255)
stroke(255, 128, 0); // bright orange (red 255, green 128, blue 0)
stroke(#FF8000); // bright orange as a web color
stroke(255, 128, 0, 128); // bright orange with 50% transparency
```

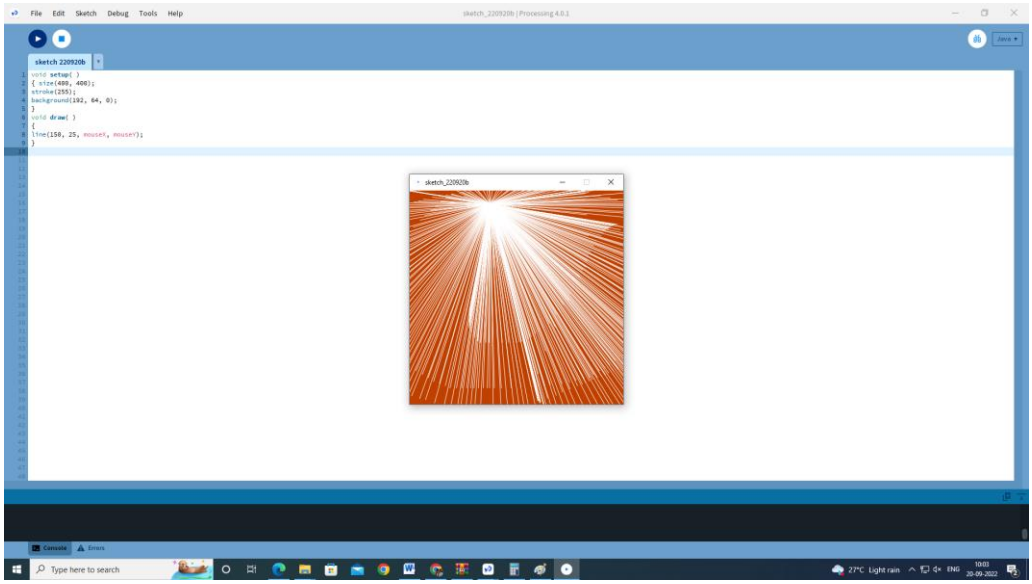
The same alternatives work for the fill( ) command, which sets the fill color, and the background( ) command, which clears the display window. Like all Processing methods that affect drawing properties, the fill and stroke colors affect all geometry drawn to the screen until the next fill and stroke commands are executed.

Hello Mouse

A program written as a list of statements is called a basic mode sketch. In basic mode, a series of commands are used to perform tasks or create a single image without any animation or interaction. Interactive programs are drawn as a series of frames, which we can create by adding functions titled setup( ) and draw( ), as shown in the continuous mode sketch in Example 2-2. They are built-in functions that are called automatically.

Example 2-2. Simple continuous mode sketch

```
void setup( )
{ size(400, 400);
stroke(255);
background(192, 64, 0);
}
void draw( )
{
line(150, 25, mouseX, mouseY);
}
```

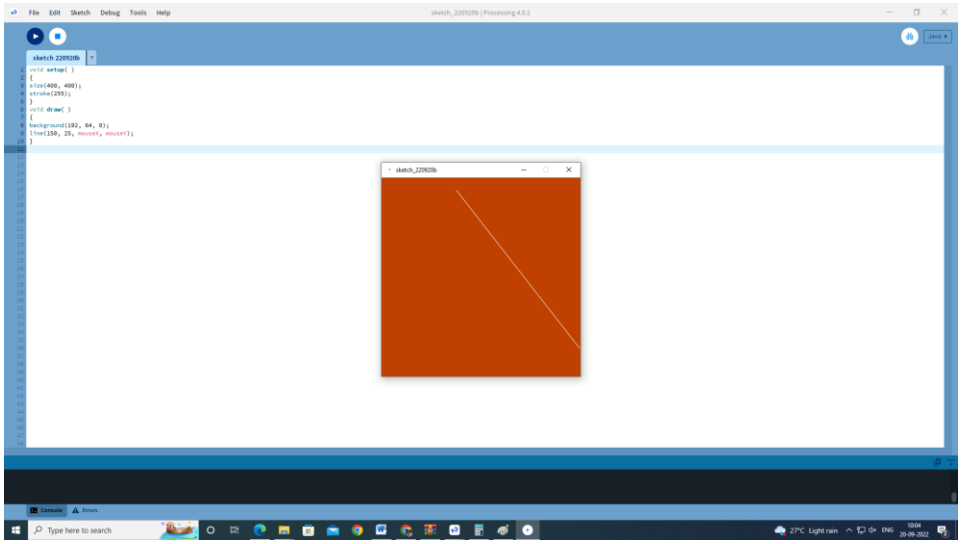


Example 2.2 is identical in function to Example 2-1, except that now the line follows the mouse. The setup( ) block runs once, and the draw( ) block runs repeatedly. As such, setup( ) can be used for any initialization; in this case, it's used for setting the screen size, making the background orange, and setting the stroke color to white. The draw( ) block is used to handle animation. The size( ) command must always be the first line inside setup( ).

Because the background( ) command is used only once, the screen will fill with lines as the mouse is moved. To draw just a single line that follows the mouse, move the background( ) command to the draw( ) function, which will clear the display window (filling it with orange) each time draw( ) runs:

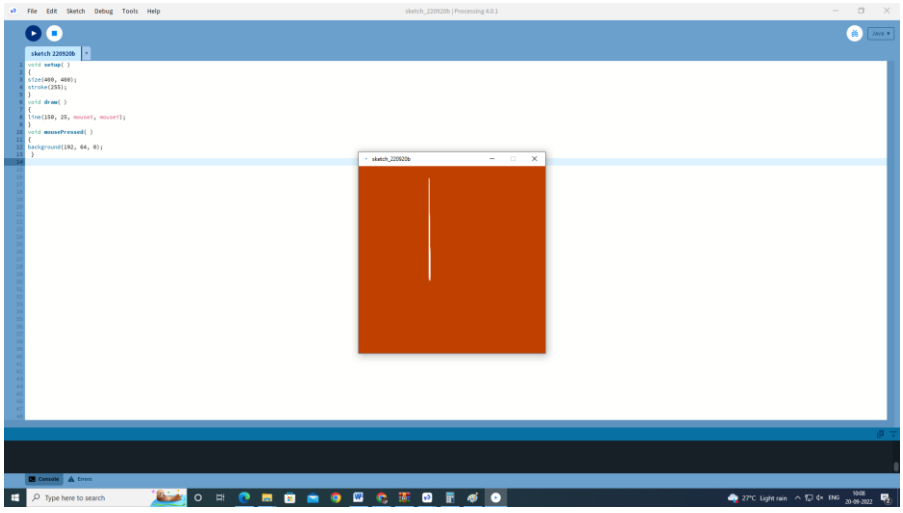
```
void setup( )
{
size(400, 400);
```

```
stroke(255);
}
void draw( )
{
background(192, 64, 0);
line(150, 25, mouseX, mouseY);
}
```



Most programs employ continuous mode, which uses the setup( ) and draw( ) blocks. More advanced mouse handling can also be introduced; for instance, the mousePressed( ) method will be called whenever the mouse is pressed. So, in the following example, when the mouse is pressed, the screen is cleared via the background( ) command:

```
void setup( )
{
size(400, 400);
stroke(255);
}
void draw( )
{
line(150, 25, mouseX, mouseY);
}
void mousePressed( )
{
background(192, 64, 0);
 }
```



Exporting and Distributing the Work

One of the most significant features of the Processing environment is its ability to bundle our sketch into an applet or application with just one click. Select File → Export to package your current sketch as an applet. This will create a folder named applet inside your sketch folder. Opening the index.html file inside that folder will open your sketch in a browser. The applet folder can be copied to a web site intact and will be viewable by users who have Java installed on their systems. Similarly, you can use File → Export Application to bundle your sketch as an application for Windows, Mac OS X, and Linux.

The applet and application folders are overwritten whenever you export—make a copy or remove them from the sketch folder before making changes to the index.html file or the contents of the folder.

Saving Your Work

If you don't want to distribute the actual project, you might want to create images of its output instead. Images are saved with the saveFrame( ) function. Adding saveFrame( ) at the end of draw( ) will produce a numbered sequence of TIFF format images of the program's output, named screen-0001.tif, screen-0002.tif, and so on. A new file will be saved each time draw( ) runs. Watch out because this can quickly fill your sketch folder with hundreds of files. You can also specify your own name and file type for the file to be saved with a command like:

```
saveFrame("output.png")
```

To do the same for a numbered sequence, use #s (hash marks) where the numbers should be placed: saveFrame("output-####.png");
For high-quality output, you can write geometry to PDF files instead of the screen.

More About the size( ) Method

The size( ) command also sets the global variables width and height. For objects whose size is dependent on the screen, always use the width and height variables instead of a number (this prevents problems when the size( ) line is altered):

```
size(400, 400);

// The wrong way to specify the middle of the screen
ellipse(200, 200, 50, 50);

// Always the middle, no matter how the size( ) line changes
ellipse(width/2, height/2, 50, 50);
```

In the earlier examples, the size( ) command specified only a width and height for the new window. An optional parameter to the size( ) method specifies how graphics are rendered. A renderer handles how the Processing API is implemented for a particular output method (whether the screen, or a screen driven by a high-end graphics card, or a PDF file). Here are examples of how to specify them with the size( ) command along with descriptions of their capabilities.

size(400, 400, JAVA2D);
The Java2D renderer is used by default, so this statement is identical to size(400, 400). The Java2D renderer does an excellent job with high-quality 2D vector graphics, but at the expense of speed. In particular, working with pixels is slower compared to the P2D and P3D renderers.

size(400, 400, P2D);
The Processing 2D renderer is intended for simpler graphics and fast pixel operations. It lacks niceties such as stroke caps and joins on thick lines, but makes up for it when you need to draw thousands of simple shapes or directly manipulate the pixels of an image or video.
size(400, 400, P3D);
Similar to P2D, the Processing 3D renderer is intended for speed and pixel operations. It also produces 3D graphics inside a web browser, even without the use of a library like Java3D. Image quality is poorer (the smooth( ) command is disabled, and image accuracy is low), but you can draw thousands of triangles very quickly.

size(400, 400, OPENGL);

The OpenGL renderer uses Sun's Java for OpenGL (JOGL) library for faster rendering, while retaining Processing's simpler graphics APIs and the PDE's easy applet and application export. To use OpenGL graphics, you must select Sketch → Import Library → OpenGL in addition to altering your size( ) command. OpenGL applets also run within a web browser without additional modification, but a dialog box will appear asking users whether they trust "Sun Microsystems, Inc." to run Java for OpenGL on their computers. If this poses a problem, the P3D renderer is a simpler, if less full-featured, solution.

size(400, 400, PDF, "output.pdf");

The PDF renderer draws all geometry to a file instead of the screen. Like the OpenGL library, you must import the PDF library before using this renderer. This is a cousin of the Java2D renderer, but instead writes directly to PDF files.

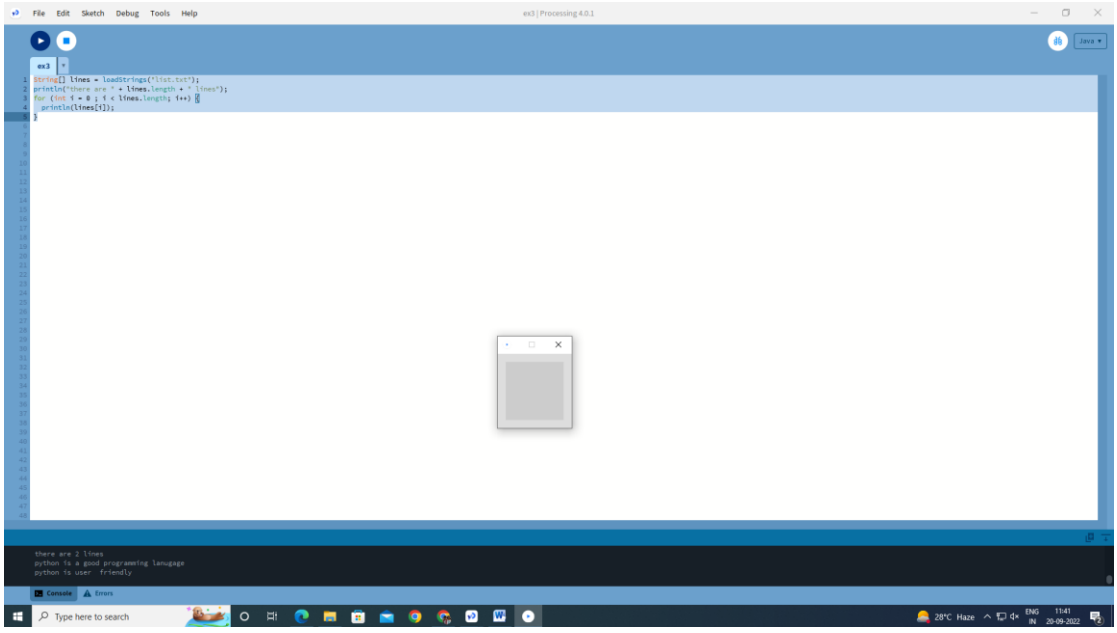
Loading and Displaying Data

One of the unique aspects of the Processing API is the way files are handled. The loadImage( ) and loadStrings( ) functions each expect to find a file inside a folder named data, which is a subdirectory of the sketch folder.

File handling functions include loadStrings( ), which reads a text file into an array of String objects, and loadImage( ), which reads an image into a PImage object, the container for image data in Processing.

Example:

```
String[] lines = loadStrings("list.txt");
println("there are " + lines.length + " lines");
for (int i = 0 ; i < lines.length; i++)
{
  println(lines[i]);
}
```

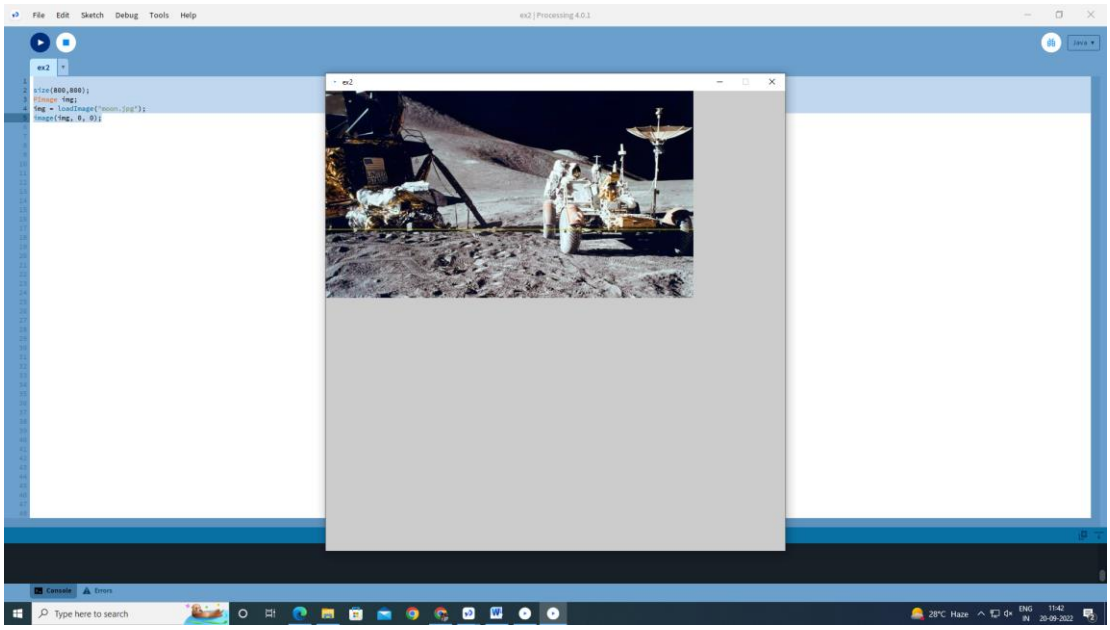


The String[] syntax means "an array of data of the class String." This array is created by the loadStrings command and is given the name lines. The reason loadStrings creates an array is that it splits the list.txt file into its individual lines.

The second command creates a single variable of class PImage, with the name image.

Example:

```
size(800,800);
PImage img;
img = loadImage("moon.jpg");
image(img, 0, 0);
```



To add a file to a Processing sketch, use the Sketch → Add File command, or drag the file into the editor window of the PDE. The data folder will be created if it does not exist already.

To view the contents of the sketch folder, use the Sketch → Show Sketch Folder command. This opens the sketch window in your operating system's file browser.

Functions

The steps of the process outlined in the first chapter are commonly associated with specific functions in the Processing API. For instance:

Acquire
    loadStrings( ), loadBytes( )
Parse
    split( )
Filter
    for( ), if (item[i].startsWith( ))
Mine
    min( ), max( ), abs( )
Represent
    map( ), beginShape( ), endShape( )
Refine
    fill( ), strokeWeight( ), smooth( )
Interact
    mouseMoved( ), mouseDragged( ), keyPressed( )

Libraries Add New Features

A library is a collection of code in a specified format that makes it easy to use within Processing. Libraries have been important to the growth of the project because they let developers make new features accessible to users without making them part of the core Processing API.

Several core libraries come with Processing. These can be seen in the Libraries section of the online reference :http://processing.org/reference/libraries.

To use the XML library in a project, choose Sketch → Import Library → xml. This will add the following line to the top of the sketch:

```
import processing.xml.*;
```

Java programmers will recognize the import command. In Processing, this line also determines what code is packaged with a sketch when it is exported as an applet or application.

Now that the XML library is imported, you can issue commands from it. For instance, the following line loads an XML file named sites.xml into a variable named xml:

```
XMLElement xml = new XMLElement(this, "sites.xml");
```

The xml variable can now be manipulated as necessary to read the contents. The full example can be seen in the reference for its class, XMLElement, at http://processing.org/reference/libraries/xml/XMLElement.html.

The this variable is used frequently with library objects because it lets the library make use of the core API functions to draw to the screen or load files. The latter case applies to the XML library, allowing XML files to be read from the data folder or other locations supported by the file API methods. Other libraries provide features such as writing QuickTime movie files, sending and receiving MIDI commands, sophisticated 3D camera control, and access to MySQL databases.

Sketching and Scripting

Processing sketches are made up of one or more tabs, with each tab representing a piece of code. The environment is designed around projects that are a few pages of code, and often three to five tabs in total. This covers a significant number of projects developed to test and prototype ideas, often before embedding them into a larger project or building a more robust application for broader deployment.

This small-scale development style is useful for data visualization in two primary scenarios. The most common scenario is when you have a data set in mind, or a question that you're trying to answer, and you need a quick way to load the data, represent it, and see what's there. This is important because it lets you take an inventory of the data in question. How many elements are there? What are the largest and smallest values? How many dimensions are we looking at?

The idea of sketching is identical to that of scripting, except that you're not working in an interpreted scripting language, but rather gaining the performance benefit of compiling to Java class files. Of course, strictly speaking, Java itself is an interpreted language, but its bytecode compilation brings it much closer to the "metal" than languages such as JavaScript, ActionScript, Python, or Ruby.

Processing was never intended as the ultimate language for visual programming; instead, we set out to make something that was:

- A sketchbook for our own work, simplifying the majority of tasks that we undertake
- A programming environment suitable for teaching programming to a non-traditional audience
- A stepping stone from scripting languages to more complicated or difficult languages such as full-blown Java or C++

MAPPING

Drawing a map is simple enough task that could be done without programming—either with mapping software or by hand—but it gives us an example upon which to build. The process of designing with data involves a great deal of iteration: small changes that help your project evolve in usefulness and clarity. And as this project evolves through the course of the chapter, it will become clear how software can be used to create representations that automatically update themselves, or how interaction can be used to provide additional layers of information.

Drawing a Map

Some development environments separate work into projects; the equivalent term for Processing is a sketch. Start a new Processing sketch by selecting File → New.

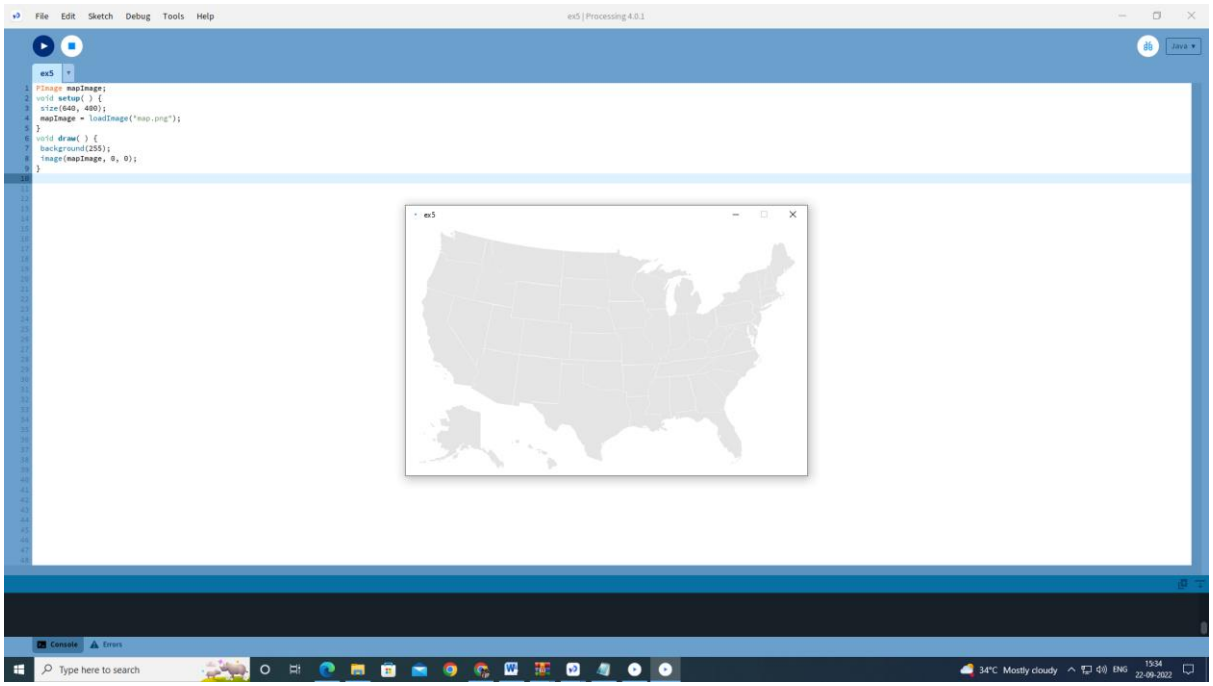
For this example, we'll use a map of the United States to use as a background image. The map can be downloaded from http://benfry.com/writing/map/map.png.

Drag and drop the map.png file into the Processing editor window. A message at the bottom will appear confirming that the file has been added to the sketch. You can also add files by selecting Sketch → Add File. A sketch is organized as a folder, and all data files are placed in a subfolder named data.

Then, enter the following code:

```
PImage mapImage;

void setup( )
{
size(640, 400);
mapImage = loadImage("map.png");
}

void draw( )
{
background(255);
image(mapImage, 0, 0);
}
```

Finally, click the Run button. Assuming everything was entered correctly, a map of the United States will appear in a new window.



Explanation of the Processing Code

Processing API functions are named to make their uses as obvious as possible. Method names, such as loadImage( ), convey the purpose of the calls in simple language. What you may need to get used to is dividing your code into functions such as setup( ) and draw( ), which determine how the code is handled.

After clicking the Run button, the setup( ) method executes once. After setup( ) has completed, the draw( ) method runs repeatedly. Use the setup( ) method to load images, fonts, and set initial values for variables. The draw( ) method runs at 60 frames per second (or slower if it takes longer than 1/60th of a second to run the code inside the draw( ) method); it can be used to update the screen to show animation or respond to mouse movement and other types of input.

Our first function calls are very basic. The loadImage( ) function reads an image from the data folder. The PImage class is a container for image data, and the image( ) command draws it to the screen at a specific location.

Locations on a Map

The next step is to specify some points on the map. To simplify this, a file containing the coordinates for the center of each state can be found at http://benfry.com/ writing/map/locations.tsv.

In future chapters, we'll explore how this data is read. In the meantime, some code to read the location data file can be found at http://benfry.com/writing/map/Table.pde.

Add both of these files to your sketch the same way that you added the map.png file earlier. The Table class is just two pages of code, and we'll get into its function later. In the meantime, suffice it to say that it reads a file as a grid of rows and columns. The class has methods to get an int, float, or String for a specific row and column. To get float values, for instance, use the following format: table.getFloat(row, column)

Rows and columns are numbered starting at zero, so the column titles (if any) will be row 0, and the row titles will be column 0.

How to display a map in Processing is a two-step process:

1. Load the data.
2. Display the data in the desired format.

Displaying the centers of states follows the same pattern, although a little more code is involved:

1. Create locationTable and use the locationTable.getFloat( ) function to read each location's coordinates (x and y values).
2. Draw a circle using those values. Because a circle, is just an ellipse whose width and height are the same, graphics libraries provide an ellipse-drawing function that covers circle drawing as well.

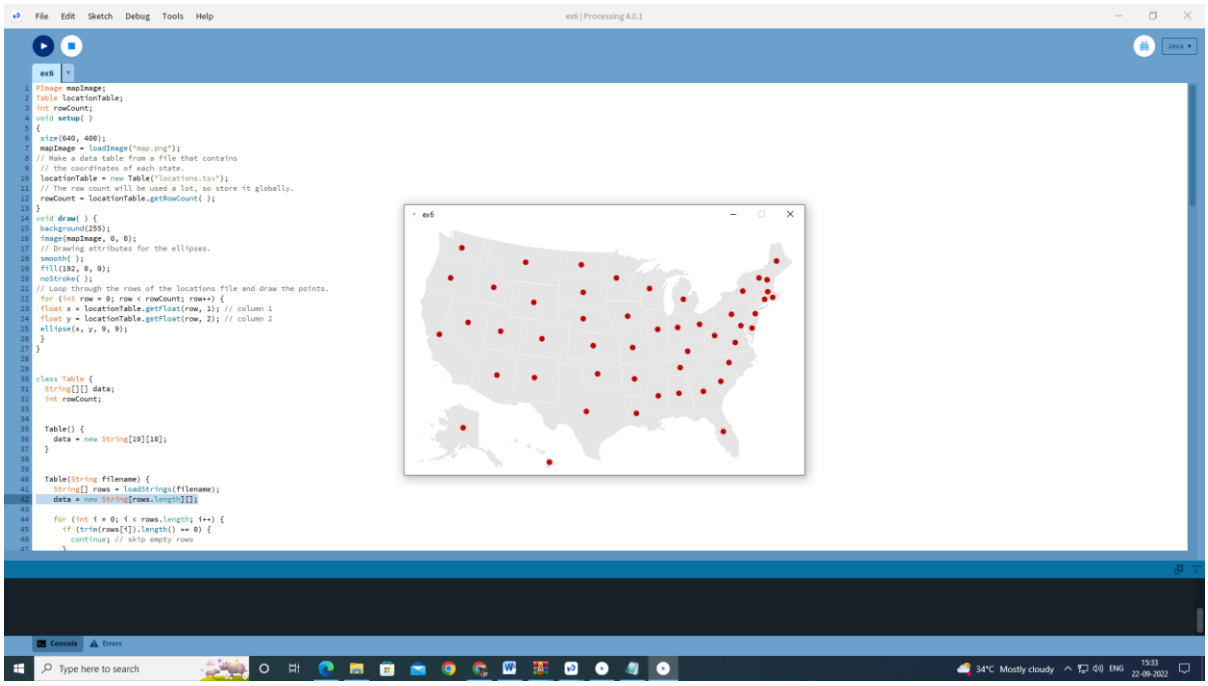
A new version of the code follows, with modifications highlighted:

```
PImage mapImage;
Table locationTable;
int rowCount;

void setup( ) {
  size(640, 400);
  mapImage = loadImage("map.png");
  // Make a data table from a file that contains
  // the coordinates of each state.
  locationTable = new Table("locations.tsv");
  // The row count will be used a lot, so store it globally.
  rowCount = locationTable.getRowCount( );
}

void draw( ) {
  background(255);
  image(mapImage, 0, 0);

  // Drawing attributes for the ellipses.
  smooth( );
  fill(192, 0, 0);
  noStroke( );

  // Loop through the rows of the locations file and draw the points.
  for (int row = 0; row < rowCount; row++) {
    float x = locationTable.getFloat(row, 1);  // column 1
    float y = locationTable.getFloat(row, 2);  // column 2
    ellipse(x, y, 9, 9);
  }
}
```

```
class Table {
 String[][] data;
 int rowCount;


 Table() {
  data = new String[10][10];
 }
```

```
Table(String filename) {
  String[] rows = loadStrings(filename);
  data = new String[rows.length][];

  for (int i = 0; i < rows.length; i++) {
    if (trim(rows[i]).length() == 0) {
      continue; // skip empty rows
    }
    if (rows[i].startsWith("#")) {
      continue;  // skip comment lines
    }

    // split the row on the tabs
    String[] pieces = split(rows[i], TAB);
    // copy to the table array
    data[rowCount] = pieces;
    rowCount++;

    // this could be done in one fell swoop via:
    //data[rowCount++] = split(rows[i], TAB);
  }
  // resize the 'data' array as necessary
  data = (String[][]) subset(data, 0, rowCount);
}

int getRowCount() {
  return rowCount;
}

  // find a row by its name, returns -1 if no row found
int getRowIndex(String name) {
  for (int i = 0; i < rowCount; i++) {
    if (data[i][0].equals(name)) {
      return i;
    }
  }
  println("No row named '" + name + "' was found");
  return -1;
}
String getRowName(int row) {
  return getString(row, 0);
}
String getString(int rowIndex, int column) {
  return data[rowIndex][column];
}

String getString(String rowName, int column) {
  return getString(getRowIndex(rowName), column);
}

  int getInt(String rowName, int column) {
  return parseInt(getString(rowName, column));
}

int getInt(int rowIndex, int column) {
  return parseInt(getString(rowIndex, column));
}

  float getFloat(String rowName, int column) {
  return parseFloat(getString(rowName, column));
```

```
  }

  float getFloat(int rowIndex, int column) {
    return parseFloat(getString(rowIndex, column));
  }


  void setRowName(int row, String what) {
    data[row][0] = what;
  }

  void setString(int rowIndex, int column, String what) {
    data[rowIndex][column] = what;
  }

  void setString(String rowName, int column, String what) {
    int rowIndex = getRowIndex(rowName);
    data[rowIndex][column] = what;
  }

    void setInt(int rowIndex, int column, int what) {
    data[rowIndex][column] = str(what);
  }
   void setInt(String rowName, int column, int what) {
    int rowIndex = getRowIndex(rowName);
    data[rowIndex][column] = str(what);
  }

  void setFloat(int rowIndex, int column, float what) {
    data[rowIndex][column] = str(what);
  }

  void setFloat(String rowName, int column, float what) {
    int rowIndex = getRowIndex(rowName);
    data[rowIndex][column] = str(what);
  }


  // Write this table as a TSV file
  void write(PrintWriter writer) {
    for (int i = 0; i < rowCount; i++) {
      for (int j = 0; j < data[i].length; j++) {
        if (j != 0) {
          writer.print(TAB);
        }
        if (data[i][j] != null) {
          writer.print(data[i][j]);
        }
      }
      writer.println();
    }
    writer.flush();
  }
}
```



Data on a Map

Next we want to load a set of values that will appear on the map itself. For this, we add another Table object and load the data from a file called random.tsv, available at http://benfry.com/writing/map/random.tsv.

It's always important to find the minimum and maximum values for the data, because that range will need to be mapped to other features (such as size or color) for display. To do this, use a for loop to walk through each line of the data table and check to see whether each value is bigger than the maximum found so far, or smaller than the minimum. To begin, the dataMin variable is set to MAX_FLOAT, a built-in value for the maximum possible float value. This ensures that dataMin will be replaced with the first value found in the table. The same is done for dataMax, by setting it to MIN_FLOAT. Using 0 instead of MIN_FLOAT and MAX_FLOAT will not work in cases where the minimum value in the data set is a positive number (e.g., 2.4) or the maximum is a negative number (e.g., –3.75).

The data table is loaded in the same fashion as the location data, and the code to find the minimum and maximum immediately follows:

```
PImage mapImage;
Table locationTable;
int rowCount;

Table dataTable;
float dataMin = MAX_FLOAT;
float dataMax = MIN_FLOAT;
void setup( )
{
 size(640, 400);
 mapImage = loadImage("map.png");
 locationTable = new Table("locations.tsv");
 rowCount = locationTable.getRowCount( );
// Read the data table.
 dataTable = new Table("random.tsv");
 // Find the minimum and maximum values.
 for (int row = 0; row < rowCount; row++)
{
 float value = dataTable.getFloat(row, 1);
 if (value > dataMax)
```

```
{
 dataMax = value;
 }
 if (value < dataMin)
{
 dataMin = value;
 }
 }
}
```

The other half of the program (shown later) draws a data point for each location. A drawData( ) function is introduced, which takes x and y coordinates as parameters, along with an abbreviation for a state. The drawData( ) function grabs the float value from column 1 based on a state abbreviation (which can be found in column 0).

The getRowName( ) function gets the name of a particular row. This is just a convenience function because the row name is usually in column 0, so it's identical to getString(row, 0). The row titles for this data set are the two-letter state abbreviations. In the modified example, getRowName( ) is used to get the state abbreviation for each row of the data file.

The getFloat( ) function can also use a row name instead of a row number, which simply matches the String supplied against the abbreviation found in column 0 of the random.tsv data file. The results are shown in Figure 3-2.

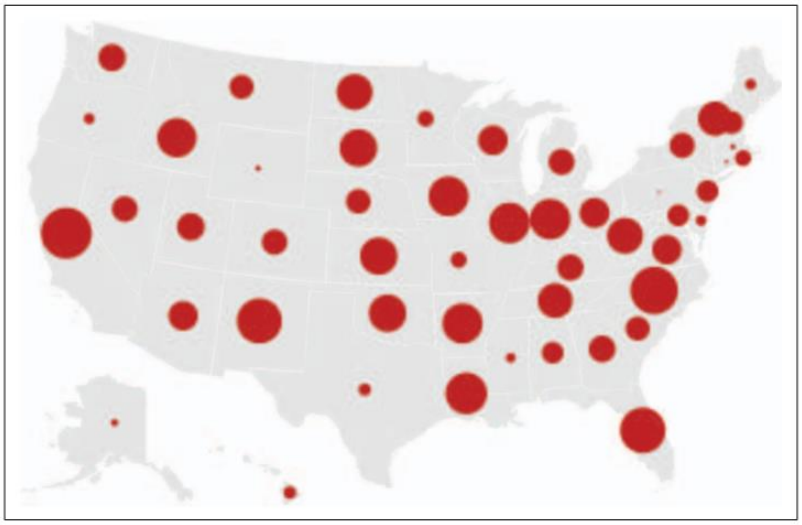


Figure 3-2. Varying data by size

The rest of the program follows:

```
void draw( )
{
background(255);
image(mapImage, 0, 0);
smooth( );
fill(192, 0, 0);
noStroke( );

for (int row = 0; row < rowCount; row++) {
 String abbrev = dataTable.getRowName(row);
 float x = locationTable.getFloat(abbrev, 1);
 float y = locationTable.getFloat(abbrev, 2);
 drawData(x, y, abbrev);
 }
}
// Map the size of the ellipse to the data value
void drawData(float x, float y, String abbrev) {
 // Get data value for state
 float value = dataTable.getFloat(abbrev, 1);
```

```
 // Re-map the value to a number between 2 and 40
 float mapped = map(value, dataMin, dataMax, 2, 40);
 // Draw an ellipse for this item
 ellipse(x, y, mapped, mapped);
}
```

The map( ) function converts numbers from one range to another. In this case, value is expected to be somewhere between dataMin and dataMax. Using map( ) reproportions value to be a number between 2 and 40. The map( ) function is useful for hiding the math involved in the conversion, which makes code quicker to write and easier to read.

Another refinement option is to keep the ellipse the same size but interpolate between two different colors for high and low values. The norm( ) function maps values from a user-specified range to a normalized range between 0.0 and 1.0. The percent value is a percentage of where value lies in the range from dataMin to dataMax. For instance, a percent value of 0.5 represents 50%, or halfway between dataMin and dataMax:

```
float percent = norm(value, dataMin, dataMax);
```

The lerp( ) function converts a normalized value to another range (norm( ) and lerp( ) together make up the map( ) function), and the lerpColor( ) function does the same, except it interpolates between two colors. The syntax:

```
color between = lerpColor(color1, color2, percent)
```

returns a between value based on the percentage (a number between 0.0 and 0.1) specified. To make the colors interpolate between red and blue for low and high values, replace the drawData( ) function with the following:

```
void drawData(float x, float y, String abbrev)
{
 float value = dataTable.getFloat(abbrev, 1);
float percent = norm(value, dataMin, dataMax);
color between = lerpColor(#FF4422, #4422CC, percent); // red to blue
fill(between);
ellipse(x, y, 15, 15);
}
```

Results are shown in Figure 3-3

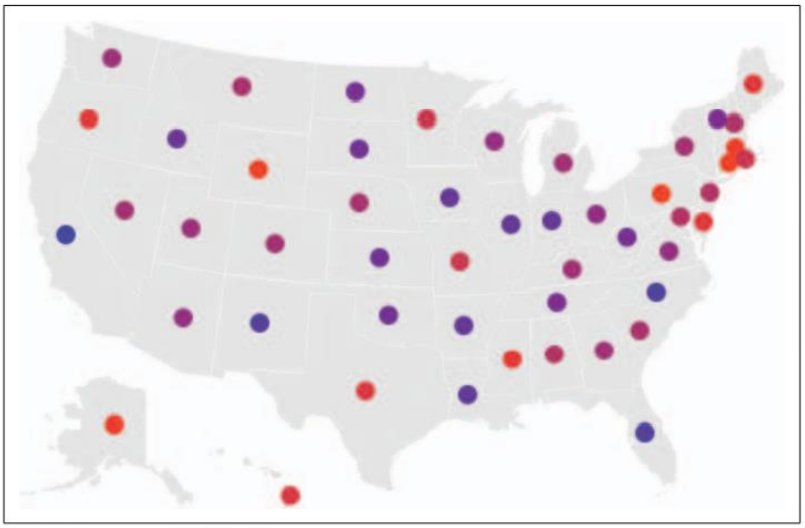


Figure 3-3. Varying data by color

TIME SERIES

The time series is a ubiquitous type of data set. It describes how some measurable feature (for instance, population, snowfall, or items sold) has changed over a period of time.

Because of its ubiquity, the time series is a good place to start when learning about visualization. With it we can cover:

- Acquiring a table of data from a text file
- Parsing the contents of the file into a usable data structure
- Calculating the boundaries of the data to facilitate representation
- Finding a suitable representation and considering alternatives

• Refining the representation with consideration for placement, type, line weight, and color
• Providing a means of interacting with the data so that we can compare variables against one another or against the average of the whole data set.

Milk, Tea, and Coffee (Acquire and Parse)

Rather than getting into the specifics of how to download and clean the data, I offer an already processed version here: http://benfry.com/writing/series/milk-tea-coffee.tsv

This data set contains three columns: the first for milk, the second for coffee, and the third for tea consumption in the United States from 1910 to 2004.

To read this file, use this modified version of the Table class from the previous chapter: http://benfry.com/writing/series/FloatTable.pde

The modified version handles data stored as float values, making it more efficient than the previous version, which simply converted the data whenever getString( ), getFloat( ), or getInt( ) were used.

Open Processing and start a new sketch. Add both files to the sketch by either dragging each into the editor window or using Sketch → Add File.

Cleaning the Table (Filter and Mine)

It's necessary to determine the minimum and maximum of each of the columns in the pre-filtered data set. These values are used to properly scale plotted points to locations on the screen. The FloatTable class has methods for calculating the min and max for the rows and columns. These methods are worth discussing because they are important in later code.

The following example calculates the minimum value for a column (comments denote important portions of the code):

```
float getColumnMax(int col)
{
// Set the value of m arbitrarily high, so the first value
// found will be set as the maximum.
float m = MIN_FLOAT;
 // Loop through each row.
for (int row = 0; row < rowCount; row++)
{
// Only consider valid data elements (see later text).
if (isValid(row, col))
{
// Finally, check to see if the value
// is greater than the maximum found so far.
if (data[row][col] > m)
{
m = data[row][col];
}
}
}
 return m;
}
```

The isValid( ) method is important because most data sets have incomplete data. In the milk-tea-coffee.tsv file, all of the data is valid, but in most data sets (including others used in this chapter), missing values require extra consideration. Because the values for milk, coffee, and tea will be compared against one another, it's necessary to calculate the maximum value across all of the columns.

The following bit of code does this after loading the milk-tea-coffee.tsv file:

```
FloatTable data;
float dataMin, dataMax;
void setup( )
{
data = new FloatTable("milk-tea-coffee.tsv");
dataMin = 0;
dataMax = data.getTableMax( );
}
```

Sometimes, it's also useful to calculate the minimum value, but setting the minimum to zero provides a more accurate comparison between the three data sets.

Each row name specifies a year, which will be used later to draw labels on the plot. To make them useful in code, it's also necessary to get the minimum and maximum year after converting the entire group to an int array. The getRowNames( ) method inside FloatTable returns a String array that can be converted with the int( ) casting function:

```
FloatTable data;
float dataMin, dataMax;
int yearMin, yearMax;
int[] years;
void setup( )
{
data = new FloatTable("milk-tea-coffee.tsv");
years = int(data.getRowNames( ));
yearMin = years[0];
yearMax = years[years.length - 1];
dataMin = 0;
dataMax = data.getTableMax( );
}
```

A Simple Plot (Represent and Refine)

To begin the representation, it's first necessary to set the boundaries for the plot location. The plotX1, plotY1, plotX2, and plotY2 variables define the corners of the plot. To provide a nice margin on the left, set plotX1 to 50, and then set the plotX2 coordinate by subtracting this value from width. This keeps the two sides even, and requires only a single change to adjust the position of both. The same technique is used for the vertical location of the plot:

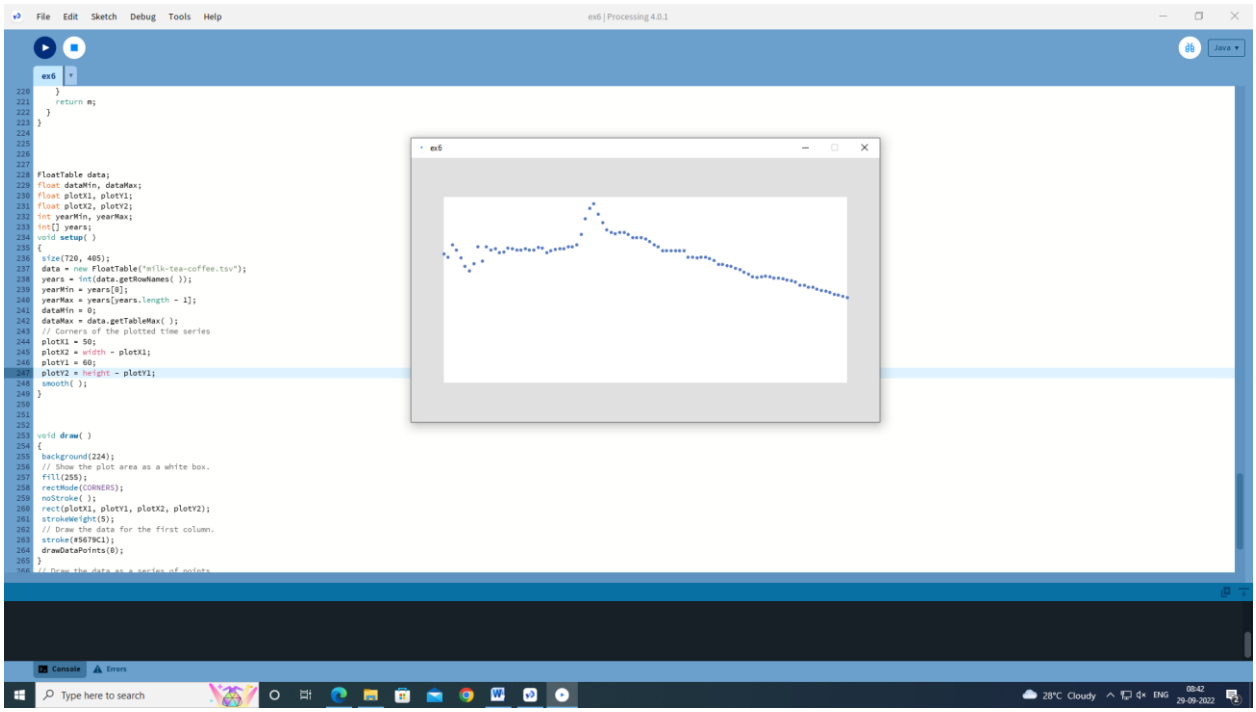
```
FloatTable data;
float dataMin, dataMax;
float plotX1, plotY1;
float plotX2, plotY2;
int yearMin, yearMax;
int[] years;
void setup( )
{
 size(720, 405);
 data = new FloatTable("milk-tea-coffee.tsv");
 years = int(data.getRowNames( ));
 yearMin = years[0];
 yearMax = years[years.length - 1];
 dataMin = 0;
 dataMax = data.getTableMax( );
 // Corners of the plotted time series
 plotX1 = 50;
 plotX2 = width - plotX1;
 plotY1 = 60;
 plotY2 = height - plotY1;
 smooth( );
}
```

Next, add a draw( ) method that sets the background to a light gray and draws a filled white rectangle for the plotting area. That will make the plot stand out against the background, rather than a color behind the plot itself—which can muddy its appearance.

The rect( ) function normally takes the form rect(x, y, width, height), but rectMode(CORNERS) changes the parameters to rect(left, top, right, bottom), which is useful because our plot's shape is defined by the corners. Like other methods that affect drawing properties, such as fill( ) and stroke( ), rectMode( ) affects all geometry that is drawn after it until the next time rectMode( ) is called:

```
void draw( )
{
 background(224);
 // Show the plot area as a white box.
 fill(255);
 rectMode(CORNERS);
 noStroke( );
 rect(plotX1, plotY1, plotX2, plotY2);
 strokeWeight(5);
 // Draw the data for the first column.
 stroke(#5679C1);
 drawDataPoints(0);
}
// Draw the data as a series of points.
void drawDataPoints(int col) {
 int rowCount = data.getRowCount( );
 for (int row = 0; row < rowCount; row++) {
 if (data.isValid(row, col)) {
 float value = data.getFloat(row, col);
 float x = map(years[row], yearMin, yearMax, plotX1, plotX2);
 float y = map(value, dataMin, dataMax, plotY2, plotY1);
 point(x, y);
 }
 }
}
```

Because the data is drawn as points using the drawDataPoints( ) method, a stroke color and weight are set. This method also takes a column index to draw as a parameter. The results are in Figure 4-1.



Trees, Hierarchies, and Recursion

Tree structures store data for which each element might have several sub-elements. Elements in a tree are typically referred to as nodes, usually with multiple child nodes. Files and directories are straightforward examples of a tree structure. Each directory can contain several items, which can be files or additional directories. Additional directories may have more files inside, and so on. This recursive structure can make trees a little tricky to deal with, but it's important because recursive structures are found in all kinds of knowledge domains.

Recursion offers special opportunities and challenges for both display and interaction:
- It's common to show one or two levels of the tree and let the user delve in or move out.
- This in turn requires ways to signal to the application that parts of the data are omitted or hidden.
- When animation is available, it's convenient to load recursive data incrementally so that you don't have to make the viewer wait for the whole data set to load.

Using Recursion to Build a Directory Tree, Start a fresh sketch, and add a second tab named Node. A Node object will be a single element in the tree. Each element will either be a file or a directory. This is typical for node structures; each is either a leaf or a container.

The node will have a (built-in) File object associated with it, which can be queried for information such as its size, last time modified, or absolute path. The following is the basic structure necessary to recursively create a tree of Node objects. As each object is created, a check is made to see whether the associated File is in fact a directory. If so, a series of child nodes are added to the children array. As those are created, the process continues: each time one of the child nodes is a directory, a series of grandchildren are added to that child node, and so on:

```
class Node {
 File file;
 Node[] children;
 int childCount;
 Node(File file) {
 this.file = file;
 if (file.isDirectory( )) {
 String[] contents = file.list( );
 children = new Node[contents.length];
 for (int i = 0 ; i < contents.length; i++) {
 File childFile = new File(file, contents[i]);
 Node child = new Node(childFile);
 children[childCount++] = child;
 }
 }
 }
}
```

Recursively Printing Tree Contents (Represent) Our initial implementation of a file tree will also offer a function to print the contents. This method is invoked by calling printList(0) on the root node. Based on the depth value, a series of spaces will be printed before the node's name. Then, it loops through the children array, calling printList( ) with the depth variable incremented by one so that each child is indented by a few more spaces.

Example 7-1 shows the final code for the Node tab. The changes from the previous example are highlighted.

Example 7-1. Recursive data structure for File objects

```
class Node {
 File file;
 Node[] children;
 int childCount;
 Node(File file) {
 this.file = file;
 if (file.isDirectory( )) {
 String[] contents = file.list( );
 if (contents != null) {
 // Sort the file names in case-insensitive order.
 contents = sort(contents);
 children = new Node[contents.length];
 for (int i = 0 ; i < contents.length; i++) {
 // Skip the . and .. directory entries on Unix systems.
 if (contents[i].equals(".") || contents[i].equals("..")) {
 continue;
 }
 File childFile = new File(file, contents[i]);
 // Skip any file that appears to be a symbolic link.
 try {
 String absPath = childFile.getAbsolutePath( );
 String canPath = childFile.getCanonicalPath( );
 if (!absPath.equals(canPath)) {
 continue;
 }
} catch (IOException e) { }
 Node child = new Node(childFile);
 children[childCount++] = child;
 }
 }
 }
 }
 void printList( ) {
 printList(0);
 }
 void printList(int depth) {
 // Print spaces for each level of depth.
 for (int i = 0; i < depth; i++) {
 print(" ");
 }
 println(file.getName( ));
 // Now handle the children, if any.
 for (int i = 0; i < childCount; i++) {
 children[i].printList(depth + 1);
 }
 }
}
void setup( )
{
 File rootFile = new File("C:\\Users\\SVIMALA.CSE\\Documents\\Processing");
 Node rootNode = new Node(rootFile);
 rootNode.printList( );
}
```

Output:

```
  lib
    classlist
    ct.sym
    fontconfig.bfc
    fontconfig.properties.src
    jawt.lib
    jfr
     default.jfc
     profile.jfc
    jrt-fs.jar
    jvm.cfg
    jvm.lib
    modules
    psfont.properties.ja
    psfontj2d.properties
    security
     blocked.certs
     cacerts
     default.policy
     public_suffix_list.dat
    tzdb.dat
    tzmappings
   release
  lib
   core.jar
   ex1.jar
   fenster.exe
   gluegen-rt.jar
   gluegen_rt.dll
   jogl-all.jar
   jogl_desktop.dll
   jogl_mobile.dll
   nativewindow_awt.dll
   nativewindow_win32.dll
   newt_head.dll
  source
   ex1.java
   ex1.pde
ex2
 data
  download.jfif
  forest.jfif
  moon.jpg
  shells.jpg
  shivan.jpeg
 ex2.pde
ex3
 data
  list.txt
 ex3.pde
 output.png
 windows-amd64
  ex3.exe
  java
    README.txt
     limited
      default_US_export.policy
      default_local.policy
      exempt_local.policy
```

Using a Queue to Load Asynchronously (Interact)

One downside of the straight recursive method used previously is that calling new Node(rootFile) won't return until it has completed. In particular, this means that for a very large directory, the program will halt completely until the entire tree structure is built. When looking at many files, this can quickly become a problem, especially in an interactive project with a draw( ) method.

Rather than create the entire tree instantaneously, a better approach is to use a queue. Each time a folder is found, it will be added to a list, and as the draw( ) method runs, a few more items from the queue can be read. This way, the program can continue without halting, and users can be updated on the progress of the files as they are read.

To implement a queue, we'll start with the code from the previous example and build it out further. The queue is handled by keeping track of a list of Node objects that have not yet been scanned for their contents in the main tab:

```
Node[] folders = new Node[10];
int folderCount;
int folderIndex;
```

The folderCount indicates the total number of folders in the list. The folderIndex variable is used to track the current folder to be read. Also in the main tab, new folders are added with the addFolder( ) method, whereas the nextFolder( ) method is used to get the next item from the queue:

```
void addFolder(Node folder) {
 if (folderCount == folders.length) {
 folders = (Node[]) expand(folders);
 }
 folders[folderCount++] = folder;
}
void nextFolder( ) {
 if (folderIndex != folderCount) {
 Node n = folders[folderIndex++];
 n.check( );
 }
}
```

The expand( ) method doubles the size of an array so that more elements can be added. The elements in the array are unaffected, and it provides an efficient means of resizing an array. It's also possible to specify a second parameter to expand( ) that indicates the new size of the array.

With this code, the nextFolder( ) method is called from draw( ) when the visualization is ready for more data. The method then calls the check( ) method (discussed shortly), which fills out the Node structure with information about child folders. Calls to the nextFolder( ) method require a negligible amount of time compared to reading several thousand files recursively and at once.

The new Node constructor uses addFolder( ) on directories, and the rest of the code from the constructor in Example 7-1 is moved to the check( ) method, called by nextFolder( ) back in the main tab:

```
Node(File file) {
this.file = file;
if (file.isDirectory( )) {
addFolder(this);
}
}
void check( ) {
String[] contents = file.list( );
if (contents != null) {
// Sort the file names in case-insensitive order.
contents = sort(contents);
children = new Node[contents.length];
for (int i = 0 ; i < contents.length; i++) {
// Skip the . and .. directory entries on Unix systems.
```

```
if (contents[i].equals(".") || contents[i].equals("..")) {
continue;
}

File childFile = new File(file, contents[i]);
// Skip any file that appears to be a symbolic link.
try {
String absPath = childFile.getAbsolutePath( );
String canPath = childFile.getCanonicalPath( );
if (!absPath.equals(canPath)) {
continue;
}
} catch (IOException e) { }
Node child = new Node(childFile);
children[childCount++] = child;
}
}
}
```

Showing Progress (Represent)

To show status information, it's necessary to set up a font for use at the end of setup( ):

```
void setup( ) {
 size(400, 130);
 File rootFile = new File("/Applications/Processing 0125");
 rootNode = new Node(rootFile);
 PFont font = createFont("SansSerif", 11);
 textFont(font);
}
```

Networks and Graphs

A graph is a collection of elements, usually called nodes, linked together by edges (sometimes called branches). It is a common structure for mapping connections of many related elements. This is partly because the visual representation of a network shows the sort of connectedness that makes sense to someone familiar with the data, whether as a free-form map of associations written out on paper (sometimes called a mind map) or, in computer science, as a visual analogue to a common data model for connections between many elements.

Porting from Java to Processing

One advantage of the move to Processing syntax is that we can hide most of the details of event handling, threading, and double buffering; all of these are handled automatically by Processing. As a result, much of the original code can be removed, simplifying the example considerably. The original code is a single .java file that contains four classes:

Node
Contains information for a single node of data

Edge
Describes a connection between two nodes, and its length

GraphPanel
The drawing surface that does most of the work

Graph
The base Applet object that handles loading a data set and starting a thread

In the Processing version, GraphPanel and Graph are merged into a single class that will be the main tab of a new sketch. Make additional tabs for the Node and Edge classes. The code in the main panel begins with a pair of arrays, one that stores Node objects and another that stores Edges:

```
int nodeCount;
Node[] nodes = new Node[100];
HashMap nodeTable = new HashMap( );
```

```
int edgeCount;
Edge[] edges = new Edge[500];
```

Next are some constants for the colors:

```
static final color nodeColor = #F0C070;
static final color selectColor = #FF3030;
static final color fixedColor = #FF8080;
static final color edgeColor = #000000;
```

The nodeColor is the yellow background for the box around each node. Clicking a node will fix it in place and change its color to fixedColor. If a node is being dragged, selectColor will be used. The edgeColor will be used to draw edges as a line between two nodes.

The setup( ) method creates the structure that will be used later for drawing, but does nothing related to the node layout on the screen; that will be handled later by the draw( ) method. setup( ) creates various parts of the structure with the assistance of a few other functions: addEdge( ), findNode( ), and addNode( ). The setup( ) code sets the size, calls the loadData( ) method, and creates a font for later use:

```
PFont font;

void setup( )
{
 size(600, 600);
 loadData( );
 font = createFont("SansSerif", 10);
 textFont(font);
 smooth( );
}
void loadData( )
{
 addEdge("joe", "food");
 addEdge("joe", "dog");
 addEdge("joe", "tea");
 addEdge("joe", "cat");
 addEdge("joe", "table");
 addEdge("table", "plate");
 addEdge("plate", "food");
 addEdge("food", "mouse");
 addEdge("food", "dog");
 addEdge("mouse", "cat");
addEdge("table", "cup");
 addEdge("cup", "tea");
 addEdge("dog", "cat");
 addEdge("cup", "spoon");
 addEdges("plate", "fork");
 addEdge("dog", "flea1");
 addEdge("dog", "flea2");
 addEdge("flea1", "flea2");
 addEdge("plate", "knife");
}
```

Adding an edge is a matter of finding each node by its name, and creating a node if it doesn't exist. Once both are found, a new Edge object is created and added to the edges array. If the array is full, it is first expanded:

```
void addEdge(String fromLabel, String toLabel) {
 Node from = findNode(fromLabel);
 Node to = findNode(toLabel);
 Edge e = new Edge(from, to);
 if (edgeCount == edges.length) {
 edges = (Edge[]) expand(edges);
```

```
}
edges[edgeCount++] = e;
```

We use expand( ) instead of Java's ArrayList or Vector classes because arrays are more runtime speed-efficient. This becomes especially important when dealing with thousands of nodes. The findNode( ) function uses a HashMap to efficiently look up a node based on its label. If none is found, addNode( ) is called:

```
Node findNode(String label) {
 label = label.toLowerCase( );
 Node n = (Node) nodeTable.get(label);
 if (n == null) {
 return addNode(label);
 }
 return n;
}
```

The addNode( ) method is much like addEdge( ), but also puts the node into the nodeTable so that it can be retrieved by name:

```
Node addNode(String label) {
 Node n = new Node(label);
 if (nodeCount == nodes.length) {
 nodes = (Node[]) expand(nodes);
 }
 nodeTable.put(label, n);
 nodes[nodeCount++] = n;
 return n;
}
```

The relax( ) methods calculate the placement of each node and lengths for each edge.

```
void draw( ) {
 background(255);
 for (int i = 0 ; i < edgeCount ; i++) {
 edges[i].relax( );
 }
 for (int i = 0; i < nodeCount; i++) {
 nodes[i].relax( );
 }
 for (int i = 0; i < nodeCount; i++) {
 nodes[i].update( );
 }
 for (int i = 0 ; i < edgeCount ; i++) {
 edges[i].draw( );
 }
 for (int i = 0 ; i < nodeCount ; i++) {
 nodes[i].draw( );
 }
}
```