12/Data

Data visualization is one of the most active areas at the intersection of code and graphics and is also among the most popular uses of Processing. This chapter builds on what's already been discussed about storing and loading data earlier in the book and introduces more features relevant to data sets that can be used for visualization.

There's a wide range of software that can output standard visualizations like bar charts and scatter plots. However, writing code to create these visualizations from scratch provides more control over the output and encourages users to imagine, explore, and create more unique representations of data. For us, this is the point of learning to code using software like Processing, and we find it far more interesting than being limited by the prepackaged methods or tools that are available.

Data Summary

It's a good time to rewind and discuss how data was introduced throughout this book. A variable in a Processing sketch is used to store a piece of data. We started with primitives. In this case, the word *primitive* means a single piece of data. For instance, an `int` stores one whole number and cannot store more than one. The idea of *data types* is essential. Each kind of data is unique and is stored in a different way. Floating-point numbers (numbers with decimals), integers (no decimals), and alphanumeric symbols (letters and numbers) all have different *types* of data to store that kind of information like `float`, `int`, and `char`.

An array is created to store a list of elements within a single variable name. For instance, Example 11-8 on page 155 stores hundreds of floating-point numbers that are used to set the stroke value of lines. Arrays can be created from any primitive data type, but they are restricted to storing a single type of data. The way to store more than one data type within a single data structure is to create a `class`.

The `String`, `PImage`, `PFont`, and `PShape` classes store more than one data element and each is unique. For instance, a `String` can store more than one character, a word, sentence, paragraph, or more. In addition, it has methods to get the length of the data or return upper- or lowercase versions of it. As another example, a `PImage` has an array called `pixels` and variables that store the width and the height of the image.

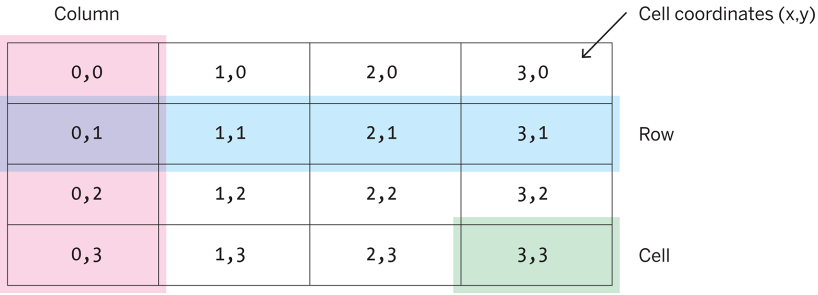
Objects created from the `String`, `PImage`, and `PShape` classes can be defined within the code, but they can also be loaded from a file within a sketch's *data* folder. The examples in this chapter will also load data into sketches, but they utilize new classes that store data in different ways.

The `Table class` is introduced for storing data in a table of rows and columns. The `JSONObject` and `JSONArray` classes are introduced to store data loaded in through files that use the *JSON* format. The file formats for `Table`, `JSONObject`, and `JSONArray` are discussed in more detail in the following section.

The *XML* data format is another native data format for Processing and it's documented in the *Processing Reference*, but it's not covered in this text.

Tables

Many data sets are stored as rows and columns, so Processing includes a `Table class` to make it easier to work with tables. If you have worked with spreadsheets, you have a head start in working with tables in code. Processing can read a table from a file, or create a new one directly in code. It's also possible to read and write to any row or column and modify individual cells within the table. In this chapter, we will focus on working with table data.



0,0	1,0	2,0	3,0
0,1	1,1	2,1	3,1
0,2	1,2	2,2	3,2
0,3	1,3	2,3	3,3

Figure 12-1. *Tables are grids of cells. Rows are the horizontal elements and columns are vertical. Data can be read from individual cells, rows, and columns.*

Table data is often stored in plain-text files with columns using commas or the tab character. A *comma-separated values* file is abbreviated as *CSV* and uses the file extension *.csv*. When tabs are used, the extension *.tsv* is sometimes used.

To load a CSV or TSV file, first place it into a sketch's *data* folder as described at the beginning of Chapter 7, and then use the `loadTable()` function to get the data and place it into an object made from the `Table` class.

Only the first few lines of each data set are shown in the following examples. If you're manually typing the code, you'll need the entire *.csv*, *.json*, or *.tsv* file to replicate the visualizations shown in the figures. You can get them from an example sketch's *data* folder (see "Examples and Reference" on page 11).

The data for the next example is a simplified version of Boston Red Sox player David Ortiz's batting statistics from 1997 to 2014. From left to right, it lists the year, number of home runs, runs batted in (RBIs), and batting average. When opened in a text editor, the first five lines of the file looks like this:

```
1997,1,6,0.327
1998,9,46,0.277
1999,0,0,0
```

```
2000,10,63,0.282
2001,18,48,0.234
```

Example 12-1: Read the Table

In order to load this data into Processing, an object from the `Table class` is created. The object in this example is called `stats`. The `loadTable()` function loads the *ortiz.csv* file from the *data* folder in the sketchbook. From there, the `for` loop reads through each table row in sequence. It gets the data from the table and saves it into `int` and `float` variables. The `getRowCount()` method is used to count the number of rows in the data file. Because this data is Ortiz's statistics from 1997 to 2014, there are 18 rows of data to read:

```
Table stats;

void setup() {
  stats = loadTable("ortiz.csv");
  for (int i = 0; i < stats.getRowCount(); i++) {
    // Gets an integer from row i, column 0 in the file
    int year = stats.getInt(i, 0);
    // Gets the integer from row i, column 1
    int homeRuns = stats.getInt(i, 1);
    int rbi = stats.getInt(i, 2);
    // Read a number that includes decimal points
    float average = stats.getFloat(i, 3);
    println(year, homeRuns, rbi, average);
  }
}
```

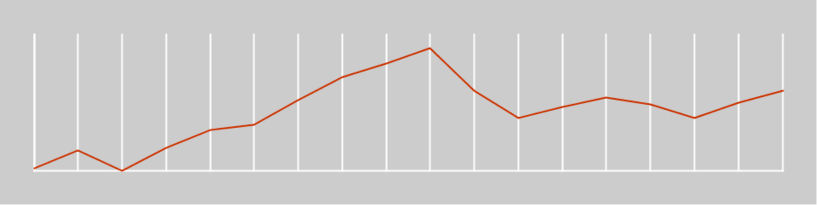
Inside the `for` loop, the `getInt()` and `getFloat()` methods are used to grab the data from the table. It's important to use the `getInt()` method to read integer data and likewise to use `getFloat()` for floating-point variables. Both of these methods have two parameters, the first is the row to read from and the second is the column.

Example 12-2: Draw the Table

The next example builds on the last. It creates an array called `homeRuns` to store data after it is loaded inside `setup()` and the data from that array is used within `draw()`. The length of homeRuns is used three times with the code homeRuns.length, to count the number of for loop iterations.

homeRuns is used first in setup() to define how many times to get an integer from the table data. Second, it is used to place a vertical mark on the graph for each data item in the array. Third, it is used to read each element of the array one by one and to stop reading from the array at the end. After the data is loaded inside setup() and read into the array, the rest of this program applies what was learned in Chapter 11.

This example is the visualization of a simplified version of Boston Red Sox player David Ortiz's batting statistics from 1997 to 2014 drawn from a table:



```
int[] homeRuns;

void setup() {
  size(480, 120);
  Table stats = loadTable("ortiz.csv");
  int rowCount = stats.getRowCount();
  homeRuns = new int[rowCount];
  for (int i = 0; i < homeRuns.length; i++) {
    homeRuns[i] = stats.getInt(i, 1);
  }
}

void draw() {
  background(204);
  // Draw background grid for data
  stroke(255);
  line(20, 100, 20, 20);
  line(20, 100, 460, 100);
  for (int i = 0; i < homeRuns.length; i++) {
    float x = map(i, 0, homeRuns.length-1, 20, 460);
    line(x, 20, x, 100);
  }
  // Draw lines based on home run data
```

```
    noFill();
    stroke(204, 51, 0);
    beginShape();
    for (int i = 0; i < homeRuns.length; i++) {
      float x = map(i, 0, homeRuns.length-1, 20, 460);
      float y = map(homeRuns[i], 0, 60, 100, 20);
      vertex(x, y);
    }
    endShape();
  }
```

This example is so minimal that it's not necessary to store this data in arrays, but the idea can be applied to more complex examples you might want to make in the future. In addition, you can see how this example could be enhanced with more information—for instance, information on the vertical axis to state the number of home runs and on the horizontal to define the year.

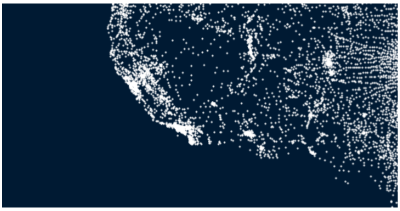
Example 12-3: 29,740 Cities

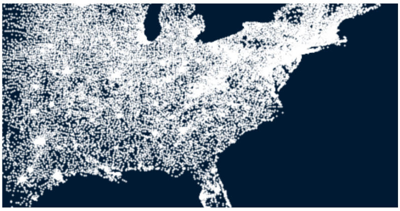
To get a better idea about the potential of working with data tables, the next example uses a larger data set and introduces a convenient feature. This table data is different because the first row, the first line in the file, is a *header*. The header defines a label for each column to clarify the context. This is the first five lines of our new data file called *cities.csv*:

```
zip,state,city,lat,lng
35004,AL,Acmar,33.584132,-86.51557
35005,AL,Adamsville,33.588437,-86.959727
35006,AL,Adger,33.434277,-87.167455
35007,AL,Keystone,33.236868,-86.812861
```

The header makes it easier to read the code—for example, the second line of the file states the zip code of Acmar, Alabama, is 35004 and defines the latitude of the city as 33.584132 and the longitude as -86.51557. In total, the file is 29,741 lines long and it defines the location and zip codes of 29,740 cities in the United States.

The next example loads this data within `setup()` and then draws it to the screen in a `for` loop within `draw()`. The `setXY()` function converts the latitude and longitude data from the file into a point on the screen:





```
Table cities;

void setup() {
  size(240, 120);
  cities = loadTable("cities.csv", "header");
  stroke(255);
}

void draw() {
  background(0, 26, 51);
  float xoffset = map(mouseX, 0, width, -width*3, -width);
  translate(xoffset, -300);
  scale(10);
  strokeWeight(0.1);
  for (int i = 0; i < cities.getRowCount(); i++) {
    float latitude = cities.getFloat(i, "lat");
    float longitude = cities.getFloat(i, "lng");
    setXY(latitude, longitude);
  }
}

void setXY(float lat, float lng) {
  float x = map(lng, -180, 180, 0, width);
  float y = map(lat, 90, -90, 0, height);
  point(x, y);
}
```

Within `setup()`, notice a second parameter `"header"` is added to `loadTable()`. If this is not done, the code will treat the first line of the CSV file as data and not as the title of each column.

The `Table class` has dozens of methods for features like adding and removing columns and rows, getting a list of unique entries in a column, or sorting the table. A more complete list of methods along with short examples is included in the *Processing Reference*.

JSON

The JSON (JavaScript Object Notation) format is another common system for storing data. Like HTML and XML formats, the elements have labels associated with them. For instance, the data for a film could include labels for the title, director, release year, rating, and more.

These labels will be paired with the data like this:

```
"title": "Alphaville"
"director": "Jean-Luc Godard"
"year": 1964
"rating": 7.2
```

To work as a JSON file, the film labels need a little more punctuation to separate the elements. Commas are used between each data pair and braces enclose it. The data defined within the curly braces is a *JSON object*.

With these changes, our valid JSON data file looks like this:

```
{
  "title": "Alphaville",
  "director": "Jean-Luc Godard",
  "year": 1964,
  "rating": 7.2
}
```

There's another interesting detail in this short JSON sample related to data types: you'll notice that the title and director data is contained within quotes to mark them as `String` data and the year and rating are without quotes to define them as numbers. Specifically, the year is an integer and the rating is a floating-point number. This distinction becomes important after the data is loaded into a sketch.

To add another film to the list, a set of brackets placed at the top and bottom are used to signify that the data is an array of JSON objects. Each object is separated by a comma.

Putting it together looks like this:

```
[
  {
    "title": "Alphaville",
    "director": "Jean-Luc Godard",
    "year": 1964,
    "rating": 7.2
  },
  {
    "title": "Pierrot le Fou",
    "director": "Jean-Luc Godard",
    "year": 1965,
    "rating": 7.7
  }
]
```

This pattern can be repeated to include more films. At this point, it's interesting to compare this JSON notation to the corresponding table representation of the same data.

As a CSV file, the data looks like this:

```
title, director, year, rating
Alphaville, Jean-Luc Godard, 1964, 9.1
Pierrot le Fou, Jean-Luc Godard, 1965, 7.7
```

Notice that the CSV notation has fewer characters, which can be important when working with massive data sets. On the other hand, the JSON version is often easier to read because each piece of data is labeled.

Now that the basics of JSON and its relation to a `Table` has been introduced, let's look at the code needed to read a JSON file into a Processing sketch.

Example 12-4: Read a JSON File

This sketch loads the JSON file seen at the beginning of this section, the file that includes only the data for the single film *Alphaville*:

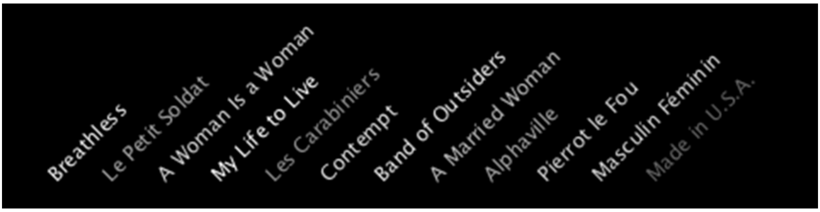
```
JSONObject film;

void setup() {
  film = loadJSONObject("film.json");
  String title = film.getString("title");
  String dir = film.getString("director");
  int year = film.getInt("year");
  float rating = film.getFloat("rating");
  println(title + " by " + dir + ", " + year);
  println("Rating: " + rating);
}
```

The `JSONObject` `class` is used to create a code object to store the data. Once that data is loaded, each individual piece of data can be read in sequence or by requesting the data related to each label. Notice that different data types are requested by the name of the data type. The `getString()` method is used for the name of the film and the `getInt()` method is used for the release year.

Example 12-5: Visualize Data from a JSON File

To work with the JSON file that includes more than one film, we need to introduce a new `class`, the `JSONArray`. Here, the data file started in the previous example has been updated to include all of the director's films from 1960–1966. The name of each film is placed in order on screen according to the release year and assigned a gray value based on the rating value.

There are several differences between this example and Example 12-4 on page 173. The most important is the way the JSON file is loaded into `Film` objects. The JSON file is loaded within `setup()` and each `JSONObject` that represents a single film is passed into the constructor of the `Film` `class`. The constructor assigns the `JSONObject` data to `String`, `float`, and `int` fields inside each `Film` object. The `Film` `class` also has a method to display the name of the film:



```
Film[] films;

void setup() {
  size(480, 120);
  JSONArray filmArray = loadJSONArray("films.json");
  films = new Film[filmArray.size()];
  for (int i = 0; i < films.length; i++) {
    JSONObject o = filmArray.getJSONObject(i);
    films[i] = new Film(o);
  }
}

void draw() {
  background(0);
  for (int i = 0; i < films.length; i++) {
    int x = i*32 + 32;
    films[i].display(x, 105);
  }
}

class Film {
  String title;
  String director;
  int year;
  float rating;

  Film(JSONObject f) {
    title = f.getString("title");
    director = f.getString("director");
    year = f.getInt("year");
    rating = f.getFloat("rating");
  }

  void display(int x, int y) {
    float ratingGray = map(rating, 6.5, 8.1, 102, 255);
    pushMatrix();
    translate(x, y);
    rotate(-QUARTER_PI);
    fill(ratingGray);
    text(title, 0, 0);
    popMatrix();
  }
}
```

This example is bare bones in its visualization of the film data. It shows how to load the data and how to draw based on those data values, but it's your challenge to format it to accentuate what you find interesting about the data. For example, is it more interesting to show the number of films Godard made each year? Is it more interesting to compare and contrast this data with the films of another director? Will all of this be easier to read with a different font, sketch size, or aspect ratio? The skills introduced in the earlier chapters in this book can be applied to bring this sketch to the next step of refinement.

Network Data and APIs

Public access to massive quantities of data collected by governments, corporations, organizations, and individuals is changing our culture, from the way we socialize to how we think about intangible ideas like privacy. This data is most often accessed through software structures called *APIs*.

The acronym *API* is mysterious and its meaning—application programming interface—isn't much clearer. However, APIs are essential for working with data and they aren't necessarily difficult to understand. Essentially, they are requests for data made to a service. When data sets are huge, it's not practical or desired to copy the entirety of the data; an API allows a programmer to request only the trickle of data that is relevant from a massive sea.

This concept can be more clearly illustrated with a hypothetical example. Let's assume there's an organization that maintains a database of temperature ranges for every city within a country. The API for this data set allows a programmer to request the high and low temperatures for any city during the month of October in 1972. In order to access this data, the request must be made through a specific line or lines of code, in the format mandated by the data service.

Some APIs are entirely public, but many require authentication, which is typically a unique user ID or key so the data service can keep track of its users. Most APIs have rules about how many, or how frequently requests can be made. For instance, it might be possible to make only 1,000 requests per month, or no more than one request per second.

Processing can request data over the Internet when the computer running the program is online. CSV, TSV, JSON, and XML files can be loaded using the corresponding load function with a

URL as the parameter. For instance, the current weather in Cincinnati is available in JSON format.

Read the URL closely to decode it:

1. It requests data from the *api* subdomain of the *openweathermap.org* site.
2. It specifies a city to search for (*q* is an abbreviation for *query*, and is frequently used in URLs that specify searches).
3. It also indicates that the data will be returned in imperial format, meaning the temperature will be in Fahrenheit. Replacing *imperial* with *metric* will provide temperature data in degrees Celsius.

Looking at this data from OpenWeatherMap is a more realistic example of working with data found in the wild rather than the simplified data sets introduced earlier. At the time of this writing, the file returned from that URL looks like this:

```
{"message":"accurate","cod":"200","count":1,"list":[{"id":
4508722,"name":"Cincinnati","coord":{"lon":-84.456886,"lat":
39.161999},"main":{"temp":34.16,"temp_min":34.16,"temp_max":
34.16,"pressure":999.98,"sea_level":1028.34,"grnd_level":
999.98,"humidity":77},"dt":1423501526,"wind":{"speed":
9.48,"deg":354.002},"sys":{"country":"US"},"clouds":{"all":
80},"weather":[{"id":803,"main":"Clouds","description":"broken
clouds","icon":"04d"}]}]}
```

This file is much easier to read when it's formatted with line breaks, and the *JSON object* and array structures defined with braces and brackets:

```
{
  "message": "accurate",
  "count": 1,
  "cod": "200",
  "list": [{
    "clouds": {"all": 80},
    "dt": 1423501526,
    "coord": {
      "lon": -84.456886,
      "lat": 39.161999
    },
    "id": 4508722,
    "wind": {
      "speed": 9.48,
      "deg": 354.002
    },
    "sys": {"country": "US"},
    "name": "Cincinnati",
    "weather": [{
      "id": 803,
      "icon": "04d",
      "description": "broken clouds",
      "main": "Clouds"
    }],
    "main": {
      "humidity": 77,
      "pressure": 999.98,
      "temp_max": 34.16,
      "sea_level": 1028.34,
      "temp_min": 34.16,
      "temp": 34.16,
      "grnd_level": 999.98
    }
  }]
}
```

Note that brackets are seen in the “`list`” and “`weather`” sections, indicating an array of *JSON objects*. Although the array in this example only contains a single item, in other cases, the API might return multiple days or variations of the data from multiple weather stations.

Example 12-6: Parsing the Weather Data

The first step in working with this data is to study it and then to write minimal code to extract the desired data. In this case, we're curious about the current temperature. We can see that our temperature data is 34.16. It's labeled as `temp` and it's inside the `main` object, which is inside the `list` array. A function called `getTemp()` was written for this example to work with the format of this specific JSON file organization:

```
void setup() {
  float temp = getTemp("cincinnati.json");
  println(temp);
}

float getTemp(String fileName) {
  JSONObject weather = loadJSONObject(fileName);
  JSONArray list = weather.getJSONArray("list");
  JSONObject item = list.getJSONObject(0);
  JSONObject main = item.getJSONObject("main");
  float temperature = main.getFloat("temp");
  return temperature;
}
```

The name of the JSON file, *cincinnati.json*, is passed into the getTemp() function inside setup() and loaded there. Next, because of the format of the JSON file, a series of JSONArray and JSONObject files are needed to get deeper and deeper into the data structure to finally arrive at our desired number. This number is stored in the temperature variable and then returned by the function to be assigned to the temp variable in setup() where it is printed to the Console.

Example 12-7: Chaining Methods

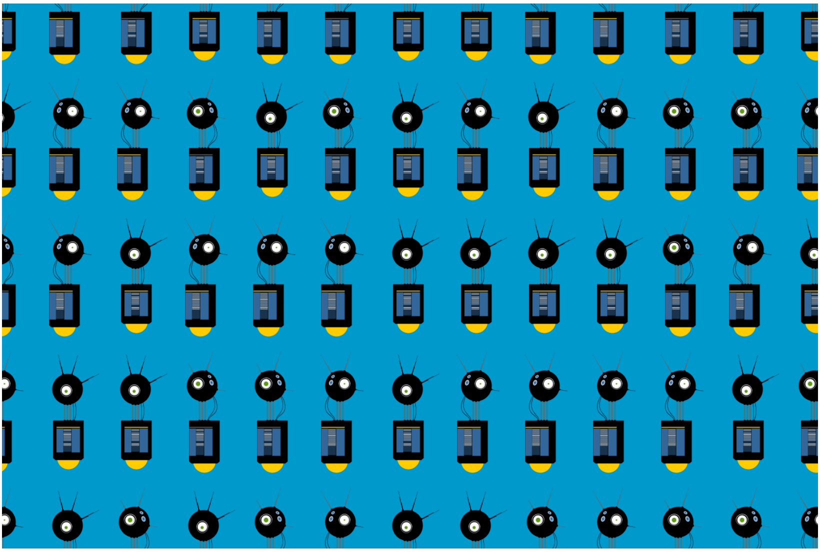
The sequence of JSON variables created in succession in the last example can be approached differently by chaining the get methods together. This example works like Example 12-6 on page 178, but the methods are connected with the dot operator, rather than calculated one at a time and assigned to objects in between:

```
void setup() {
  float temp = getTemp("cincinnati.json");
  println(temp);
}

float getTemp(String fileName) {
  JSONObject weather = loadJSONObject(fileName);
  return weather.getJSONArray("list").getJSONObject(0).
  getJSONObject("main").getFloat("temp");
}
```

Also note how the final temperature value is returned by the getTemp() function. In Example 12-6 on page 178, a float variable is created to store the decimal value, then that value is returned. Here, the data created by the get methods is returned directly, without intermediate variables.

This example can be modified to access more of the data from the feed and to build a sketch that displays the data to the screen rather than just writing it to the Console. You can also modify it to read data from another online API—you'll find that the data returned by many APIs shares a similar format.

Robot 10: Data



The final robot example in this book is different from the rest because it has two parts. The first part generates a data file using random values and `for` loops and the second part reads that data file to draw an army of robots onto the screen.

The first sketch uses two new code elements, the `PrintWriter class` and the `createWriter()` function. Used together, they create and open a file in the *sketchbook* folder to store the data generated by the sketch. In this example, the object created from `PrintWriter` is called `output` and the file is called *botArmy.tsv*. In the loops, data is written into the file by running the `println()` method on the output object. Here, random values are used to define which of three robot images will be drawn for each coordinate. For the file to be correctly created, the `flush()` and `close()` methods must be run before the program is stopped.