Second Edition

Processing

A Programming Handbook for Visual Designers and Artists

Casey Reas

Ben Fry

Foreword by John Maeda

This book was set in TheSerif and TheSansMono from LucasFonts by Toppan Best-set Premedia Limited. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication

Reas, Casey.
Processing : a programming handbook for visual designers and artists / Casey Reas and Ben Fry.—Second edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-262-02828-8 (hardcover : alk. paper) 1. Computer programming. 2. Computer graphics—Computer programs. 3. Digital art—Computer programs. 4. Art—Data processing. 5. Art and technology. I. Fry, Ben. II. Title.

QA76.6.R4138 2014
006.60285—dc23 2014016283

10 9 8 7 6 5 4 3 2

32 Data

This chapter introduces data formatting and file writing and reading.

Syntax introduced:

```
nf(), saveStrings(), exit()
PrintWriter, createWriter(), PrintWriter.flush(), PrintWriter.close()
loadStrings(), split()
Table, loadTable(), Table.getRowCount(), Table.getInt(),
Table.getString(), TableRow, Table.rows()
XML, loadXML(), XML.getChildren(), XML.getContent(), XML.getString()
JSONArray, loadJSONArray(), JSONArray.size()
JSONObject, JSONArray.getJSONObject(), JSONObject.getString(),
JSONObject.getInt(), JSONObject.getFloat()
```

Digital files are not tangible like their paper namesakes. Despite the diverse content stored in digital files, the material of each is the same—a sequence of 1s and 0s stored on a hard drive or other media. Almost every task performed with computers involves working with files. Editing a single text file involves first reading from dozens of other files that comprise the text editor application itself. When the document is saved, the file is given a name and stored for later retrieval.

The primary reason to save a file is to store data so that it is available after a program stops running. When it is running, a program uses part of the computer's memory to store its data temporarily. When the program is stopped, the program gives control of this memory back to the operating system so other programs can access it. If the data created by the program is not saved to a file, it is lost when the program closes.

All software files have a *file format*, a convention for ordering data so that software applications know how to interpret the data when it is read from memory. Formats are often referred to by their *extension*, usually three or four letters following a period at the end of the file's name. Some common extensions include TXT for plain text files, MP3 for storing sound, and EXE for executable programs on Windows. Common formats for image data include JPEG and GIF and formats for documents include ODT and DOC. The XML and JSON formats have become popular in recent years as general-purpose data formats that can be extended to hold specific types of data in an easy-to-read file.

Format data

Text files contain characters that are not visible (referred to as nonprintable) and are used to define the spacing of the visible characters. The two most common are *tab* and *new line*. These characters can be represented in code as \t and \n, respectively. The combination of the \ (backslash) character with another is called an *escape sequence*. These escape sequences are treated as one character by the computer. The backslash begins the escape sequence and the second character defines the meaning. A new line is helpful for controlling layout to make a file easier to read, while using a tab to delimit text in a file can make it easier to load it back into a program and separate pieces of data.

```
// Prints "tab	space"
println("tab\tspace");
```
32-01

```
// Prints each word after "\n" on a new line:
// line1
// line2
// line3
println("line1\nline2\nline3");
```
32-02

Data can also be formatted with functions such as `nf()` to control the number of digits in a number. (The function name `nf()` is short for *number format*.) In addition to formatting the data, the `nf()` function converts the data into the `String` type so it can be output to the console or saved to a text file. This function has two or three parameters to control how a number is formatted. The first parameter is always the number to format, an `int`, `float`, or array of numbers. When one additional parameter is used, it adds zeros before the number to set the total number of digits. For example:

```
println(nf(200, 10));          // Prints "0000000200"
println(nf(40, 5));            // Prints "00040"
println(nf(90, 3));            // Prints "090"
```
32-03

When two additional parameters are used with `nf()`, the first sets the number of digits to the left of the decimal, and the second sets the number of digits after the decimal. Setting either parameter to zero means "any" number of digits.

```
println(nf(200.94, 10, 4));    // Prints "0000000200.9400"
println(nf(40.2, 5, 3));       // Prints "00040.200"
println(nf(9.012, 0, 5));      // Prints "9.01200"
```
32-04

Export files

Saving files is a useful way to store data so it can be viewed after a program has stopped running. Data can either be saved continuously while the program runs or stored in variables while the program is running; and then it can be saved to a file in one batch.

The `saveStrings()` function writes an array of strings to a file, with each string written to a new line. This file is saved to the sketch's folder and can be accessed by selecting the "Show Sketch Folder" item from the Sketch menu. The following example uses the `saveStrings()` function to write data created while drawing lines to the screen. Each time a mouse button is pressed, a new value is added to the `x[]` and `y[]` arrays, and when a key is pressed the data stored in these arrays is written to a file called *lines.txt*. The `exit()` function then stops the program.

```
int[] x = new int[0];
int[] y = new int[0];

void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  stroke(0);
  noFill();
  beginShape();
  for (int i = 0; i < x.length; i++) {
    vertex(x[i], y[i]);
  }
  endShape();
  // Show the next segment to be added
  if (x.length >= 1) {
    stroke(255);
    line(mouseX, mouseY, x[x.length-1], y[x.length-1]);
  }
}

void mousePressed() {  // Click to add a line segment
  x = append(x, mouseX);
  y = append(y, mouseY);
}

void keyPressed() {  // Press a key to save the data
  String[] lines = new String[x.length];
  for (int i = 0; i < x.length; i++) {
    lines[i] = x[i] + "\t" + y[i];
  }
  saveStrings("lines.txt", lines);
  exit();  // Stop the program
}
```

The `PrintWriter` class provides another way to export files. Instead of writing the entire file at one time as `saveStrings()` does, the `createWriter()` method opens a file to write to and allows data to be added continuously to the file while the program is running. To make the file save correctly, the `flush()` method must be used to write any remaining data to the file. The `close()` method is also needed to finish writing the file properly. The following example uses the `PrintWriter` to save the cursor position to a file while a mouse button is pressed.

32-06

```
PrintWriter output;

void setup() {
  size(100, 100);
  // Create a new file in the sketch directory
  output = createWriter("positions.txt");
  frameRate(12);
}

void draw() {
  if (mousePressed) {
    point(mouseX, mouseY);
    // Write the coordinate to a file with a
    // "\t" (TAB character) between each entry
    output.println(mouseX + "\t" + mouseY);
  }
}

void keyPressed() {      // Press a key to save the data
  output.flush();        // Write the remaining data
  output.close();        // Finish the file
  exit();                // Stop the program
}
```

The file created with the previous program has a simple format. The x-coordinate of the cursor is written followed by a tab, then followed by the y-coordinate. Code 32-08 shows how to load this file back into another sketch and use the data to redraw the saved points.

The next example is a variation of the previous one, but it uses the spacebar and Enter key to control when data is written to the file and when the file is closed. When a key is pressed, the character is added to the `letters` variable. When the spacebar is pressed, the `String` is written to the *words.txt* file. When the Enter key is pressed, the file is flushed and then closed, and the program exits.

32-07

32-06

```
String letters = "";
PrintWriter output;

void setup() {
  size(100, 100);
  fill(0);
  // Create a new file in the sketch folder
  output = createWriter("words.txt");
}

void draw() {
  background(204);
  text(letters, 5, 50);
}

void keyPressed() {
  if (key == ' ') {               // Spacebar pressed
    output.println(letters);  // Write data to words.txt
    letters = "";                 // Clear the letter String
  } else {
    letters = letters + key;
  }
  if ((key == ENTER) || (key == RETURN)) {
    output.flush();               // Write the remaining data
    output.close();               // Finish the file
    exit();                       // Stop the program
  }
}
```

Data structure

Files are the easiest way to store and load data, but before you load a data file into a program, it is essential to know how the file is formatted. In a plain text file, the control characters for tab and new line are used to differentiate and align the data elements. Separating the individual elements with a tab or space character and each line with a new line character is a common formatting technique. Here is one example excerpted from a data file:

```
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
```

If you see a file formatted in a similar way, you can use a text editor to tell whether there are tabs or spaces between the elements by moving the cursor to the beginning of a line and using the arrow keys to navigate left or right through the characters. If the cursor jumps from one element to another, there is a tab between the elements; if the cursor moves via a series of steps through the whitespace, spaces were used to format the data. In addition to knowing how the data elements are separated, it's essential to know how many data elements each line contains and the data type of each element. For example, the file above has data that should be stored as `String`, `int`, and `float` variables.

In addition to loading data from plain text files, it's common to load data from XML and JSON files. Both are file structures based on *tagging* information. Each defines a structure for ordering data, but leaves the content and categories of the data elements open. For example, in an XML structure designed for storing book information, each element might have an entry for title and publisher:

```
<book>
    <title>Processing</title>
    <publisher>MIT Press</publisher>
</book>
```

In an XML structure designed for storing a list of websites, each element might have an entry for the name of the website and the URL.

```
<website>
    <name>Processing.org</name>
    <url>http://processing.org</url>
</website>
```

In these two examples, notice that the names of the element tags are different, but the structure is the same. Each entry is defined with a tag to begin the data and a corresponding tag to end the entry. Because the tag for each data element describes the type of content, XML files are often more self-explanatory than files delimited by tabs. Processing can load and parse simple, strictly formatted XML files.

JSON files work similarly to XML files, but the notation is different. The data in JSON files, the acronym is for JavaScript Object Notation, is defined by braces and

colons, rathe
translated to

```
{
  "title":
  "publish
}

{
    "name":
    "url":
}
```

Tab-delimite
available fro
instance, wea
websites. In t
for these con
file is more a
considerable
lightweight t
the excerpt p
comprises se
necessary or

Strings

The basic way
to extract the
characters, wl
loadString
individual lin
into Processin
text file *numb*
Processing wi

```
  String[
```

The `lines[]`
`loadString`
array contain
the array and

```
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
```

If you see a file formatted in a similar way, you can use a text editor to tell whether there are tabs or spaces between the elements by moving the cursor to the beginning of a line and using the arrow keys to navigate left or right through the characters. If the cursor jumps from one element to another, there is a tab between the elements; if the cursor moves via a series of steps through the whitespace, spaces were used to format the data. In addition to knowing how the data elements are separated, it's essential to know how many data elements each line contains and the data type of each element. For example, the file above has data that should be stored as `String`, `int`, and `float` variables.

In addition to loading data from plain text files, it's common to load data from XML and JSON files. Both are file structures based on *tagging* information. Each defines a structure for ordering data, but leaves the content and categories of the data elements open. For example, in an XML structure designed for storing book information, each element might have an entry for title and publisher:

```
<book>
    <title>Processing</title>
    <publisher>MIT Press</publisher>
</book>
```

In an XML structure designed for storing a list of websites, each element might have an entry for the name of the website and the URL.

```
<website>
    <name>Processing.org</name>
    <url>http://processing.org</url>
</website>
```

In these two examples, notice that the names of the element tags are different, but the structure is the same. Each entry is defined with a tag to begin the data and a corresponding tag to end the entry. Because the tag for each data element describes the type of content, XML files are often more self-explanatory than files delimited by tabs. Processing can load and parse simple, strictly formatted XML files.

JSON files work similarly to XML files, but the notation is different. The data in JSON files, the acronym is for JavaScript Object Notation, is defined by braces and

colons, rathe
translated to

```
{
  "title":
  "publish
}

{
    "name":
    "url": "
}
```

Tab-delimite
available fro
instance, wea
websites. In t
for these con
file is more a
considerable
lightweight t
the excerpt p
comprises sev
necessary or

Strings

The basic way
to extract the
characters, wh
loadString
individual lin
into Processin
text file *numb*
Processing wi

```
  String[
```

The lines[]
loadStrings
array containi
the array and

colons, rather than with angle brackets. The two XML examples above may be translated to JSON as follows:

```
{
    "title": "Processing",
    "publisher": "MIT Press"
}
{
    "name": "Processing.org",
    "url": "http://processing.org"
}
```

Tab-delimited, XML, and JSON data are each useful in different contexts. Many "feeds" available from the web are available in XML and JSON format. These include, for instance, weather service updates from the NOAA and the RSS feeds common to many websites. In these cases, the data is both varied and hierarchical, making it suitable for these configurations. For information exported from a database, a tab-delimited file is more appropriate, because the additional metadata included in XML wastes considerable space and takes longer to load into a program. JSON is generally more lightweight than XML, but it's still extra markup that is often not needed. For example, the excerpt presented above is from a file that contains 40,000 lines. Because the data comprises seven straightforward columns, adding additional tags or structure is not necessary or desired.

Strings

The basic way to bring external data into Processing is to load and parse a TXT file to extract the individual data elements. A TXT file format stores only plain text characters, which means there is no formatting such as bold, italics, and colors. The `loadStrings()` function reads the contents of a file and creates a string array of its individual lines—one array element for each line of the file. As with any media loaded into Processing, the file must be located in the sketch's data folder. For example, if the text file *numbers.txt* is in the current sketch's data folder, its data can be read into Processing with this line of code:

```
String[] lines = loadStrings("numbers.txt");
```

The `lines[]` array is first declared and then assigned the `String` array created by the `loadStrings()` function. It holds the contents of the file, with each element in the array containing one line of the text in the file. This code reads through each element of the array and prints its contents to the Processing console:

colons, rather than with angle brackets. The two XML examples above may be translated to JSON as follows:

```
{
  "title": "Processing",
  "publisher": "MIT Press"
}
{
  "name": "Processing.org",
  "url": "http://processing.org"
}
```

Tab-delimited, XML, and JSON data are each useful in different contexts. Many "feeds" available from the web are available in XML and JSON format. These include, for instance, weather service updates from the NOAA and the RSS feeds common to many websites. In these cases, the data is both varied and hierarchical, making it suitable for these configurations. For information exported from a database, a tab-delimited file is more appropriate, because the additional metadata included in XML wastes considerable space and takes longer to load into a program. JSON is generally more lightweight than XML, but it's still extra markup that is often not needed. For example, the excerpt presented above is from a file that contains 40,000 lines. Because the data comprises seven straightforward columns, adding additional tags or structure is not necessary or desired.

Strings

The basic way to bring external data into Processing is to load and parse a TXT file to extract the individual data elements. A TXT file format stores only plain text characters, which means there is no formatting such as bold, italics, and colors. The `loadStrings()` function reads the contents of a file and creates a string array of its individual lines—one array element for each line of the file. As with any media loaded into Processing, the file must be located in the sketch's data folder. For example, if the text file *numbers.txt* is in the current sketch's data folder, its data can be read into Processing with this line of code:

```
String[] lines = loadStrings("numbers.txt");
```

The `lines[]` array is first declared and then assigned the `String` array created by the `loadStrings()` function. It holds the contents of the file, with each element in the array containing one line of the text in the file. This code reads through each element of the array and prints its contents to the Processing console:

```
  for (int i = 0; i < lines.length; i++) {
    println(lines[i]);
  }
```

The following example loads the text file created with code 32-06. This file contains the `mouseX` and `mouseY` variable separated by a tab and formatted like this:

```
x1      y1
x2      y2
x3      y3
x4      y4
x5      y5
```

The program is designed to read the entire file into an array; then it reads each line of the array and extracts the two coordinate values into another array. The file checks to make sure the data is formatted as expected by confirming that there are two elements on each line, then converts these elements to integer values and uses them to draw a point to the screen.

32-08

```
String[] lines;
int index = 0;

void setup() {
  size(100, 100);
  lines = loadStrings("positions.txt");
  noSmooth();
  frameRate(12);
}

void draw() {
  if (index < lines.length) {
    String[] pieces = split(lines[index], '\t');
    if (pieces.length == 2) {
      int x = int(pieces[0]);
      int y = int(pieces[1]);
      point(x, y);
    }
    // Go to the next line for the next run through draw()
    index++;
  }
}
```

The `split()` function is used to divide each line of the text file into its separate elements. This function splits a string into pieces using a character or string as the

divider. The
The next pa
char or Str
example to

```
String s
String[] p
println(p
println(p
```

Table

The coordin
with Proces
columns. It
to working
as simple as
32-08 to util

```
Table xy;
int index

void setup
  size(10
  xy = lc
  noSmooth
  frameRa
}

void draw(
  if (ind
    int x
    int y
      point
    // Go
    index
  }
}
```

The loadTa
table structu
data format.
to define "ta

divider. The first parameter to the function is the string to separate into smaller pieces. The next parameter is the data delimiter; frequently a comma, tab, or space. It can be a `char` or `String` and does not appear in the returned `String[]` array. A simplified example to show how it works follows:

32-09

```
String s = "a, b";
String[] p = split(s, ", ");
println(p[0]);  // Prints "a"
println(p[1]);  // Prints "b"
```

Table

The coordinate data loaded as strings in code 32-08 can be loaded in a different way with Processing's `Table` class. The `Table` class stores data in a matrix of rows and columns. It has methods to access and modify this data. The `Table` class is well suited to working with files that combine multiple types of data, but it can also manage data as simple as a series of x- and y-coordinates. The following example translates code 32-08 to utilize a table.

32-10

```
Table xy;
int index = 0;

void setup() {
  size(100, 100);
  xy = loadTable("positions.txt", "tsv");
  noSmooth();
  frameRate(12);
}

void draw() {
  if (index < xy.getRowCount()) {
    int x = xy.getInt(index, 0);
    int y = xy.getInt(index, 1);
    point(x, y);
    // Go to the next line for the next run through draw()
    index++;
  }
}
```

The `loadTable()` function is used to put the data from the *positions.txt* file into the table structure. The second parameter to `loadTable()` tells the function about the data format. In this case, the data is separated by tabs, so the parameter `"tsv"` is used to define "tab separated values." (Note that if the file name ends with .tsv, it's not

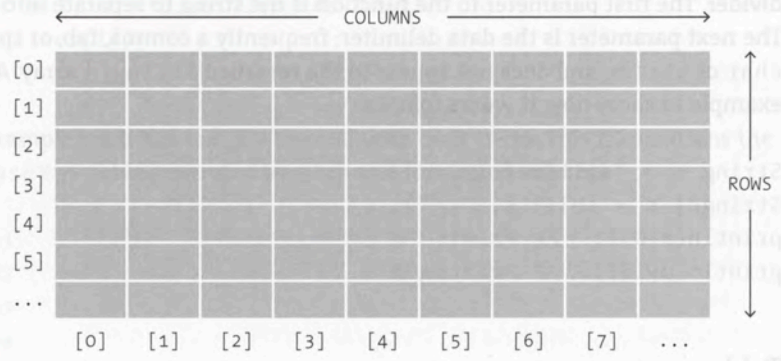

Figure 32-1 Table
A table is a data matrix defined by rows that run top to bottom and columns that run left to right. Each data element has a location defined by its row and column.

necessary to add the second parameter.) Once this is finished, the data can be read, changed, added to, and deleted through the many Table class methods. These methods are listed in detail in the Processing reference. This example uses the `getRowCount()` method as well as `getInt()`. The `getRowCount()` and its companion, `getColumnCount()` methods return the dimensions of the table. In the case of this data, there are two columns, one to store the x-coordinate data and one to store the y-coordinate data. There are 206 rows because that is the number of lines in the file. Next, the `getInt()` method returns the data at the location defined by the parameters as an integer.

Files often contain multiple kinds of data, and these more complicated files are also handled well with the Table class. It's important to know what kind of data is inside a file so that it can be parsed into variables of the appropriate type. The next example loads data about automobiles to demonstrate more features of working with tables. In the file used for this example, text data is mixed with numbers:

```
ford galaxie 500    15   8   429   198   4341   10    70   1
chevrolet impala    14   8   454   220   4354   9     70   1
plymouth fury iii   14   8   440   215   4312   8.5   70   1
pontiac catalina    14   8   455   225   4425   10    70   1
```

This small excerpt of the file shows its content and formatting. Each element is separated with a tab and corresponds to a different aspect of each car. This file stores the miles per gallon, cylinders, displacement, and like information for more than 400 different cars. The next example loads the data into a table, uses a `for` loop to read the name and miles per gallon for each car, then prints that data to the console. The

enhanced f
minimal sy

```
Table car

void setu
  size(10
  cars =
  for (Ta
    Strir
    int r
    print
  }
}
```

The new th
TableRow
method to r
program th
allows the c
example, th

```
    Strir
```

rather than
is reference

The Tab
removeRow
addColumn
and matchR
Processing r

XML

As reference
brackets. No
brackets (e.g
bracket (e.g.
tag. They are
<tag />). T
appropriatel
most import
This relation

enhanced for loop is used to access each table row in order from first to last with a minimal syntax.

```
Table cars;                                                             32-11

void setup() {
  size(100, 100);
  cars = loadTable("cars.tsv", "header");
  for (TableRow row : cars.rows()) {
    String id = row.getString("name");
    int mpg = row.getInt("mpg");
    println(id + ", " + mpg + " miles per gallon");
  }
}
```

The new things in this example are the "header" parameter with loadTable(), the TableRow class and rows() methods within the for loop, and the getString() method to read text content from the file. The "header" parameter is used to tell the program that the first row in the data file lists the categories for each column. This allows the columns to be referenced by a string rather than with a number. For example, the seventh line in code 32-11 can also be written like this:

```
    String id = row.getString(0);
```

rather than the preferred manner in the example. The code is clearer when the column is referenced as "name," and less prone to errors if the columns are reordered in the file.

The Table class has additional methods to remove rows and columns, removeRow() and removeColumn(), and to add new data, addRow() and addColumn(). Table data can also be searched with methods such as findRow() and matchRow(). Examples for these, and more Table methods, are provided in the Processing reference.

XML

As referenced earlier, XML files structure data within a set of tags defined by angle brackets. Notice how each tag is repeated twice. The first time placed within clean angle brackets (e.g. <tag>) and the second time with a forward slash following the first bracket (e.g. </tag>). The first set of brackets is the start tag and the second is the end tag. They are always used as a pair, except in the case of an empty-element tag (e.g. <tag />). The name of the tags can change with each XML file to model the content appropriately, but the structure of the tags applies in the same way to all XML files. The most important idea about XML structure is the way tags are nested inside other tags. This relationship is referred to as *parent* and *child* nodes. If one tag has another