CS 106 Winter 2016
Craig S. Kaplan

Module 02
Input and Output

Topics

- Reading and displaying images and illustrations
- Writing images, illustrations, and animations
- Reading text
- Writing text

Readings

- *Learning Processing*, Sections 15.1, 18.3, 18.4, 21.3, 21.4

Introduction

So far, nearly everything you've done in Processing has been self-contained. To create practical tools, we need ways to exchange information with the outside world.

Most computer systems are backed by a *filesystem*, a place where files can be stored more permanently. Modern filesystems are very large and very complicated. Here's what my computer says about the size and number of files in my home directory:

▼ General:
Size: 272,495,822,559 bytes (274.89 GB on disk) for 335,119 items
Where: Macintosh HD ▸ Users

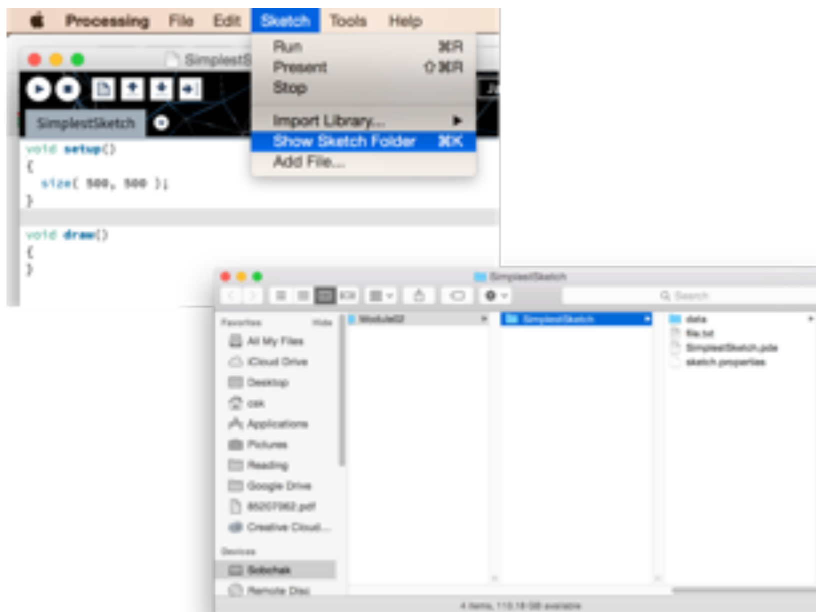
There are other reasons why filesystems are complex:

- They're usually *heterogeneous*: your files can be stored across multiple physical devices, including "in the cloud".

- The organization of your files (and even their names) depends very heavily on your operating system. Mac is very different from Windows, and iOS is very different from Android, and it's annoying to write code that knows about all these differences.

This complexity is antithetical to the spirit of Processing. So Processing offers us a way of hiding most of the complexity when programming, by having your sketch folder serve as a gateway to the outside world. By default, when you ask to read a file, Processing looks for that file in your sketch folder; when you write a file, it puts the output in your sketch folder. The easiest way to access your sketch folder is from within the Processing environment:

It's possible to ask Processing to open any file at all, if you know the file's "absolute path". We won't need to do that in this course. Usually, it's also possible to pass in a URL instead of a filename, in which case the file will be downloaded from the web. We'll probably see that later in the course.



Reading images

At the end of CS 105 you learned how to load images into Processing. First you need to move or copy the image into your sketch folder. That's most easily done using the "Add File..." command in the "Sketch" Processing menu.

"Add File..." will actually put files in a "data" subfolder of the sketch folder, creating that folder if necessary. In fact, it doesn't matter: Processing will look for files in both the main sketch folder and the data subfolder, if it exists.

Use the built-in function `loadImage()`, passing in a `String` with the name of an image file as input, to read an image into a sketch. You'll get back an object of type `PImage`. If the filename doesn't exist, you'll get back `null` and Processing will print a warning message to the Console. But the program won't stop—you can keep doing things, as long as you don't try to draw the `null` image! See the **ExtraExtra** sketch in Lab 00 for an example of this.

The most common practice is to define a global variable of type `PImage` and set that variable to the result of loading an image in your `setup()` function. Then, you can use the built-in `image()` function to display an image at any position in your sketch, optionally rescaling it to fit any rectangle. See the reference documentation for `PImage` for information on other things you can do with images (notably, find out an image's width and height, and read its pixels one-by-one).

If you call `loadImage()` in `draw()` instead, most likely everything will continue to work, but your program will be slower because you're re-reading the image from the filesystem every frame. Most of the time that's not what you want!

Example sketch: TintGrid

Reading illustrations

If you've used illustration software like Adobe Illustrator, you know that vector graphic images are very important in art and design, in addition to plain old pixels. It would be nice to have a way to import vector illustrations directly into Processing, without having to convert them into raster images (i.e., pixels) first.

Fortunately, this turns out to be as easy as working with raster images. In fact, most of the time you can simply replace "image" with "shape" in the previously mentioned functions for dealing with images:

`PImage` ⟺ `PShape`
`loadImage()` ⟺ `loadShape()`
`image()` ⟺ `shape()`

Processing's SVG support is good but incomplete. It's a good idea to stick to the most common parts of the SVG standard.

Processing definitely doesn't understand embedded CSS used to style SVG paths. When saving from Illustrator, I had to set "CSS Properties" to "Style Attributes" under advanced SVG saving options in order to make sure it didn't use any CSS.

It turns out that `loadShape()` can also be used to load 3D meshes in the Wavefront OBJ format! 3D is much more complicated to work with, but I might get to it later in the term.

Processing uses SVG as its native vector illustration format. If you have an illustration in another format (PS, EPS, PDF, etc.), it's fairly easy to find software to convert to SVG. See the reference documentation for `PShape` for more information on working with illustrations. I can imagine that the `disableStyle()` method might occasionally be useful.

Example sketch: DisplaySVG

Example sketch: Moustachify

Writing images

It's incredibly easy to save a "screenshot" of a running sketch, i.e., the current contents of the sketch window. Just use the built-in function `save()`, passing in the name of the file you wish to save to. I recommend saving to images in PNG format.

```
void keyPressed()
{
  if( key == 's' ) {
    save( "screen.png" );
  }
}
```

You could add this code fragment to just about any sketch to add in a handy screenshot feature (assuming it doesn't already have a `keyPressed()` function).

You can also use the built-in function `saveFrame()`, which chooses sequential filenames for you. This could be useful for stitching saved images together into an animation using heavy-duty software like Adobe Premiere, or a simple tool like the "Movie Maker' built in to Processing.

Writing illustrations

So far we can read and write images, and read illustrations. We should fill in the obvious gap, and learn how to create new vector illustrations from within a Processing sketch.

Happily, this turns out to be very easy in Processing, using the PDF library that ships with Processing. This library is stored “off to the side” in Processing—you must explicitly ask for the features of the library to be brought into your sketch, otherwise they won’t be available. To do so, we use an *import directive*. The directive for the PDF library looks like this:

```
import processing.pdf.*;
```

An import directive is a feature that Processing inherited from Java. It finds all the declarations in an external library, and makes them available in the current sketch. Without the `import`, your sketch won’t have access to the features of the library, even if you know they’re out there somewhere. Most libraries include documentation that tell you what you need to import.

Once you’ve got the PDF library imported, a line like

```
beginRecord( PDF, "output.pdf" );
```

Is more or less all you need to get started. This line of code causes all subsequent drawing functions (e.g., `line()`, `ellipse()`) to silently draw a second copy of themselves in the output file. And unlike the sketch window, these versions are true vector graphic elements: pure, scalable geometry. The only other step is to stop recording when you’re done drawing:

```
endRecord();
```

Here, at last, is a great way to incorporate Processing into a graphic design pipeline: write a sketch that solves some particular design problem, and load the output into software like Adobe Illustrator for further editing. I have created similar programs many times as part of my research.

Of course, you probably *don't* want to start and stop recording to an external PDF in every frame of a sketch. In practice you should either record a single time and then stop the sketch, or else have a way to record specifically when requested by the user (for example, when a key is pressed). That requires just a bit of extra code in most sketches.

Example sketch: RecordPDF

As of Processing 3.0, it's possible to save SVG files in addition to PDF. The process is nearly identical to the one above. Just use

```
import processing.svg.*;
```

And substitute `SVG` for `PDF` when calling the `beginRecord()` function.

Writing text

So far, all the reading and writing we've been doing have been based on well-known *file formats* (e.g. JPEG, GIF, SVG, PDF), and we have relied on other code to serve as an intermediary in making sense of files in those formats for us. Sometimes, though, we have information we want to save for later for which there is no specific format. In those cases, it's usually

simplest to write out plain text. Fortunately, this is easy in Processing as well—it looks a lot like the familiar `println()` function.

As you know, `println()` sends its output to the Console window underneath your sketch's source code. It's possible to create other objects that can receive `println()` messages:

```
void setup()
{
  PrintWriter writer = createWriter( "output.txt" );
  writer.println( "This will go directly into the file." );
  writer.println( "So will the number below:" );
  writer.println( 3.1415926 );
  println( "But this will still be printed to the Console." );
  writer.flush();
  writer.close();
  exit();
}
```

A `PrintWriter` is an object that understands `println()`, but that sends its output directly to a file. Note that you need to say `writer.println()` for a particular object `writer` to send text to that writer's file. If you continue to use regular `println()`, text will go to the Console, as before. In fact you could have many `PrintWriter` objects active at once, all writing to their own files.

When you're done writing, you need to call the `flush()` method to make sure the PrintWriter isn't holding any part of your file in memory instead of on disk, and then call `close()` to tell Processing you're done with the file.

Reading text

Yes, there is a `createReader()` function that fits nicely with `createWriter()`. But I'm not going to teach that approach. The problem is that in order to read files that way, you need to be aware of *exception handling*

(i.e., `try` and `catch`), an advanced programming technique.

Fortunately, we can skip that messiness with the much simpler built-in function `loadStrings()`. You call `loadStrings()` with the name of the file to be read, and get back an array of `Strings`, each of which contains exactly one line from the file. It's up to you to decide how to interpret those strings.

If it matters, reading and writing of text files is actually done in UTF-8 encoding, so you can safely use Unicode if you want.

Applications of input/output

Here are just a few sample uses of input/output.

- **Logging:** Sometimes, it isn't enough to write debugging information or other status messages to the Processing Console. The information can scroll too fast and be lost, or there could simply be too much of it to read comfortably in that tiny window. It can sometimes be useful to generate a huge torrent of information into an external file, which you can then examine later at your leisure. This can be a valuable way to find bugs in complex programs, since you have a detailed record of what happened. Many software tools you use regularly leave behind logs, even if you never see them. In OSX, have a look in "Library/Logs" inside your home directory.
- **Persistence:** One use of persistence is to keep track of enough of a running program's state that you can jump back into that state if a program is stopped and restarted. Before a program halts, you write the current state out to a file. At startup, you check if that file exists and use it to reconstruct the state if it does. This is a common idiom. On mobile devices, you can save energy by terminating apps when the user isn't actively interacting with them. Persistence allows those apps to come back to life seamlessly.
- **Data visualization:** If you generate data from some other source, Processing can be a good way to load that data and create novel visualizations of it. Later we'll see how to read CSV (comma-separated

values) files, which would allow you to draw your own custom charts and graphics based on Excel spreadsheets.