


```

<?I?H°<4Δe"...:bG€û!-êN ˘€ΔmC«çs+-?fi_çb«°$ßÛ4?Ô?ú0ÒÈ ∂
nÖ¥#ÄW◇Óÿ?iÕ±H:∂#o“CEØøh{ª?JuLGCÉÅ?
Í-tfjDZXg~?ÁFÃ k»Ê'∂Ó?∂?∂PK?∂?∂!«-°∂f∂
∂_rels/.rels ç?∂(†∂“í;N√ 0ÜÔH°C%o°ön Ñ-“]∂“ni∂0â∂∂/$J<ÿfiû∞√
J•QMb<fCEüÔ~ÔÂj∂w,ib≤fi)∂∂%r/∂Î∂∂Ør“i-Dbt∂;ÔH;û∂“™Í'Â
u»~QjmH"∏∏$†e∂R&»Rè©•Å\Æl|ÿëÛ162†,¿Ü%oç,Ôd, ≠ ∂'HS“çç∏∏67 Í}»?G[˘fhêQj-
i∂b&ãl≥∂Qclà∂∂ØüÛu:t∂ô∂%oi†€ÛÅ,fc5=zΩÌ...Ò
    œívLCEêôF-∂¶àÊó$∂3∂Û—»<$Éí)ö≈~4∂/√ ∂∂ Σ € .Õ° ì∂ícT«ZÒ-@~
    Lé÷≤'∂~∂PK?∂?∂?∂!hu· ∂π∂∂∂word/_rels/document.xml.rels ç?∂(†∂“iÕN√
0∂ÑÔH°ÉÂ;qR† Tß∂@Í∂Ç8ªCE:±àì»^~Ú^òT•âZ-∂%∂Kª+œ|kçWÎ/"ê∂•A;Àiñ$îÄiÆ' ∂
, Û·x°∏∏•$†∞·hú∂N;tùüü ≠ û † ∂∂/ÖZ Σ ÁD∂∂8 ≠ ∂€;ΔÇ“;jàè∏∂lú(Áç¿X'äµBæâ
ÿ"móÄ∂5h> “$óíSø)/)) ∫ 6:˘Øÿi “∂Óù|7^ÒÑ∂
Ä∂7
QS⁻
ê” }'áúùF∏∏ö∂·∂ ∂ œG∂ÉÊ∂»iú Y,f ∂ Å√c, ∂ ¶ ñsB` †/Y∂fS

```

The moral of this little story is that binary files have a lot of information that we are not supposed to see that is used to display the information that we are supposed to see.

Text files are WYSIWYG – What You See Is What You Get. Many programmers refer to text files as “human readable” and looking at the stuff above you, can see why this description is appropriate.

Reading a Text File

Processing has a function that does the work of reading the file:

```
loadStrings ( "name-of-the-text-file-to-read" );
```

If you want to see what this function does when it is called, look at the last section of these notes. That information is not necessary to understand and use **loadStrings ()**. It is there for the curious among you.

The simplest way to use **loadStrings ()** is to put the text file in a folder named **data** that is in the folder containing the .pde file – where you may have put your image files, sound files and font files.

The argument of the function call **loadStrings ()** must itself be a String that is the name of the file. (*If the file is located some place other than the data folder, you can put the complete path to the file – putting the file in the data folder, if at all possible, is much simpler.*).

The function **loadStrings ()** returns the information in the file as an array of Strings where each String element of the array is one line of the text file.

If the text file is empty (*yes, there can be empty text files – suppose you have a game that is keeping track of high scores in a text file. Before anyone plays your game, would be no high scores*), the array returned by `loadStrings()` will have a length of zero.

If there is one line in the text file, the returned array of Strings will have one element. If there are 42 lines in the text file the returned array will have 42 elements.

Each String (if there are any) will be one complete line of the text file.

Here is a text file named **DataDemo0.txt**:

```
how now brown cow
she sells sea shells by the sea shore
rubber baby buggy bumpers
four wrists wear four watches
peter piper picked a peck of pickled peppers
```

Here is the code that can read the file and store it in an array of Strings:

```
String [ ] dataStrings;
void setup( )
{
  size( 400, 600 );
  dataStrings = loadStrings( "DemoData0.txt" );
  println( dataStrings );
}
```

The `text()` function cannot display an array without a loop to traverse the array but the `println()` function can. Here is the output of the `println()`:

```
[0] "how now brown cow"
[1] "she sells sea shells by the sea shore"
[2] "rubber baby buggy bumpers"
[3] "four wrists wear four watches"
[4] "peter piper picked a peck of pickled peppers"
```

As you can see, the array of Strings returned by `loadStrings()` and referenced by `dataStrings` has five elements and each element is a String. The beginning element of the array is the beginning line of text in the file. The last element of the array is the last line of the file.

In our work with Processing thus far most of the data we use are numbers – locations and sizes of figures and RGB color values. “How now brown cow” is not of much use. This is where file reading gets interesting. Beyond saying use the function `loadStrings()` to read the file, what we do with the returned array of Strings can be an infinite set of possibilities.

We can show you some examples here and these examples along with Shiffmans' chapter can only serve as a guide. You have to modify the code to read your files.

First, let's suppose we have a file that has the kind of figure, location, size and RGB values of figures. The file might look like this:

```
0 100 100 40 50 200 100 45
1 100 200 33 83 100 200 0
1 200 200 73 23 50 100 200
0 300 100 76 23 50 200 55
```

Some programmers call this a formatted file. That means that each line is the same as every other line – not the same values but what the values represent. Every value is an int value. This is important if we want to keep the file reading simple. It also means that we need to know what the values represent.

Question – what does the leading 0 or 1 stand for? The answer is not obvious – it comes from the dark recesses of Jim's mind (brrr...). He is using the 0 to represent a rectangle and the 1 to represent an ellipse². So here is what the data translates to:

Rect/ellipse	X	Y	Width	Height	Red	Blue	Green
0	100	100	40	50	200	100	45
1	100	200	33	83	100	100	0
1	200	200	73	23	50	100	200
0	300	100	76	23	50	200	55

Here is the code that reads the file:

```
String [ ] dataStrings;
void setup( )
{
  size( 400, 600 );
  dataStrings = loadStrings( "DemoData1.txt" );
  println( dataStrings );
}
```

Look familiar—is should – the only change is the name of the file. Here is the output of the `println()`:

```
[0] "0 100 100 40 50 200 100 45"
[1] "1 100 200 33 83 100 200 0"
[2] "1 200 200 73 23 50 100 200"
[3] "0 300 100 76 23 50 200 55"
```

Now we have to convert this array of Strings into int values that we can use to draw the figures. If you have not read Shiffman, you should – he does this much better than Jim does.

Converting this data requires two steps.

² How could he store data for a triangle?

1. Split each String into a new array that contains the separate Strings.
2. Convert these separate Strings into int values

Let's look at #1 first. The code to split each String looks like this:

<pre>String [] dataStrings; void setup() { size(400, 600); dataStrings = loadStrings("DemoData1.txt"); for(int i = 0; i < dataStrings.length; i++) { String [] individualStrings = split(dataStrings[i], ' '); println("Array element [" + i + "]:"); println(individualStrings); } }</pre>	<p>The for loop traverses the array dataStrings</p> <p>Each element of the array which is one entire line of the file is split using the function split() which returns an array containing the individual substrings.</p> <p>This is the magic that separates each different substring in the array into separate Strings. The signature of <code>split()</code> is: <code>split(String [], char)</code> where the char is the char that separates each substring.</p> <p>This print the label for each array element.</p> <p>This prints the array of substrings..</p>
--	--

Here is the output from the `println()` for each call of `split()` ;

<pre>Array element [0]: [0] "0" [1] "100" [2] "100" [3] "40" [4] "50" [5] "200" [6] "100" [7] "45" Array element [1]: [0] "1" [1] "100" [2] "200" [3] "33" [4] "83" [5] "100" [6] "200" [7] "0" Array element [2]: [0] "1" [1] "200"</pre>

```
[2] "200"  
[3] "73"  
[4] "23"  
[5] "50"  
[6] "100"  
[7] "200"  
Array element [ 3]:  
[0] "0"  
[1] "300"  
[2] "100"  
[3] "76"  
[4] "23"  
[5] "50"  
[6] "200"  
[7] "55"
```

Ok, that takes care of the first task and we have an array of Strings but we cannot use Strings as arguments for locating and sizing rects and ellipses and the RGB values of colors. We have to convert these Strings to int values (or float values if we need to use floats).

So on to task #2 – converting the Strings to ints...

Again, Processing does the dirty work for us with the function `int()` or, if we are using floats, the function `float()`.

We have used `int()` to convert float values to int values earlier in the course. It turns out that the `int()` function has multiple signatures – something the new version of the Processing API does not show you (Jim has sent mail to the Processing folks...):

The signatures for the `int()` function appear by testing with code to be:

```
int( float )  
int( String )  
int( String [ ] )
```

So the `int()` function can convert a single String to an int value and return it as long as the String forms a valid integer AND it can convert an array of Strings into an array of int values and return the array as long as all of the Strings being converted can form valid integers.

The code to do this is below. Remember that we know what each value on each line represents. We have to know this. If you have forgotten, go back to page 4. The program now has seven parallel arrays to store the data – one array for each value on a line of the file.

<pre>String [] dataStrings; int [] fig, xLoc, yLoc, wd, ht, redVal, greenVal, blueVal; void setup() { size(400, 600); dataStrings = loadStrings("DemoData1.txt"); fig = new int [dataStrings.length]; xLoc = new int int [dataStrings.length]; yLoc = new int int [dataStrings.length]; wd = new int int [dataStrings.length]; ht = new int int [dataStrings.length]; redVal = new int int [dataStrings.length]; greenVal = new int int [dataStrings.length]; blueVal = new int int [dataStrings.length]; for(int i = 0; i < dataStrings.length; i++) { String [] individualStrings = split(dataStrings[i], ' '); int [] intData = int(individualStrings); fig[i] = intData[0]; xLoc[i] = intData[1]; yLoc[i] = intData[2]; wd[i] = intData[3]; ht[i] = intData[4]; redVal[i] = intData[5]; greenVal[i] = intData[6]; blueVal[i] = intData[7]; } }</pre>	<p>Declares the seven parallel arrays</p> <p>Read the file - when this is done, the length of dataStrings tells us the length required for each parallel array which we use when we new the seven parallel arrays.</p> <p>Here the seven parallel arrays are new'ed using the length of the dataStrings array.</p> <p>When this is finished we have the seven arrays of in that we need to store the data in the original text file.</p> <p>We traverse the dataStrings array so. . .</p> <p>We can splint each array element into substrings using the ' ' char as the separating char.</p> <p>We use the int() function to convert the array of substrings int an array of int and return it and assign it to the int array, intData. The array intData contains the seven int values on one line of the file.</p> <p>Since we know what each value represents we assign the value in intData to the [i] element of each parallel array.</p>
---	---

More Files

Files can contain data in any format that a programmer desires. The previous example was just one. We will look at two more formats and from there, you are on your own if you need or want to store data in a file. It is not difficult; it just takes some thinking and planning.

Sometimes we get data out of a spread sheet or on the web in a format that is not what we want to use. Suppose we had data in a spreadsheet that had the high temperatures for every day of the year. The spreadsheet has twelve lines; one for each month. Each line has between 28 and 31 int values for the high temperature of each day. Here is a fake version of what the spread sheet might look like – remember the numbers are faked for this discussion.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Jan	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	21	19	20	21	19	20	21
Feb	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	21	19	20	23			
Mar	49	21	45	23	26	12	31	49	21	45	23	26	12	31	23	26	12	31	49	21	45	23	26	12	31	23	26	23	26	23	26
Apr	35	33	38	32	42	41	38	32	28	43	35	33	38	32	42	41	38	32	28	43	35	33	38	32	42	41	38	32	28	43	
May	39	34	39	43	45	48	42	51	48	53	49	56	39	34	39	43	45	48	42	51	48	53	49	56	48	53	49	56	53	49	56
Jun	65	61	59	71	64	67	64	59	68	67	72	73	76	77	69	64	58	68	73	72	74	68	66	74	77	78	68	72	69	73	
Jul	74	73	77	82	84	81	79	82	68	72	77	82	85	80	83	78	85	82	80	79	77	82	83	81	88	89	92	89	91	93	97
Aug	78	77	81	86	88	85	83	86	72	76	81	86	89	84	87	82	89	86	84	83	81	86	87	85	92	93	96	93	95	97	101
Sep	82	81	85	90	92	89	87	90	76	80	85	90	93	88	91	86	93	90	88	87	85	83	78	85	82	80	79	77	82	80	82
Oct	64	60	58	60	63	66	63	58	67	66	64	62	59	56	61	63	57	67	62	61	63	67	65	63	61	59	57	58	58	56	55
Nov	35	33	38	32	42	41	38	32	28	43	35	33	38	32	42	41	38	32	28	43	35	33	38	32	42	41	38	32	28	43	
Dec	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	20	21	19	21	19	20	21	19	20	21