Exercise: XML and Flash
ActionScript 3.0 and Adobe Flash Professional

We will start with a collection of assets that are very typical: text and images.

- The images are all JPG files. They are in a folder named *images*. They have already been edited in Photoshop, so they are all consistently sized.
- The text is an MS Word document. It includes information about each one of 12 items, including the filename of an image, a book title, and an ISBN (a universally used unique ID number for published books).

Let's get the most annoying part out of the way first. It is tedious but very useful.

Data (XML) conversion sequence

We begin with a normal MS Word file with different kinds of info *on separate lines*.

1. Open the MS Word file that includes all the information about all the books (*books.docx*).[1]
2. Select all the text in the MS Word file.
3. Convert the text to a table. (If you don't know how, look in Word Help. It is clearly explained there.) When prompted, change the NUMBER OF COLUMNS to "9." Why? Because you have *eight pieces of information* about each book, *plus a blank line* after each set. (In database terms, we would say we have nine *fields* in each *record*.) The default for "Separate text at" should be *paragraphs*; that is correct (each *line* is a paragraph in Word). Click OK.
4. Select the final column in the table. All of the cells are empty (these were the blank lines). **Delete** it (select it, then right-click: Delete Columns).
5. There should be no blank cells in this table. *Check this*.
6. Select the entire table in Word. Copy it.
7. Open MS Excel. Click on the top left cell in the grid to select it. **Paste**.
8. Now we need to do a little data cleaning.[2] The reason: Many ISBNs *start* with zero. If we had pasted normal ISBNs into Excel, it would have eliminated all those leading zeros, and then you would have to type them all back in. *No one would do that.* So, the ISBNs in the document all had an apostrophe added as the first character; this is a clever trick to make Excel accept them as text. However, now we must delete the apostrophes. This is a cool Excel procedure for a common data-cleaning operation.

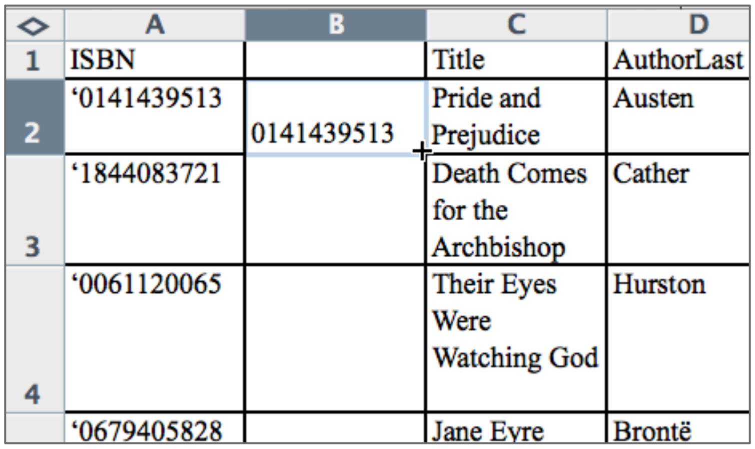
[1] Word is used (instead of a plain-text editor) because it's easy to make proper typographic quotation marks, em-dashes, accented characters, etc., in Word. Plus, everyone can understand Word.

[2] All the steps under No. 8 are specific to the ISBNs, which are numerals that need to be treated as text. *If you did not have this kind of data in the dataset,* you could go directly to No. 9 to continue with the XML conversion. Data often need some kind of cleaning to work properly in Web applications.

a. Insert *a new column* in Excel between columns A and B. The new one will now be B.
b. Click the cell in column B, row 2, to select it.
c. Paste the following text in cell B2, and then press Return or Enter.

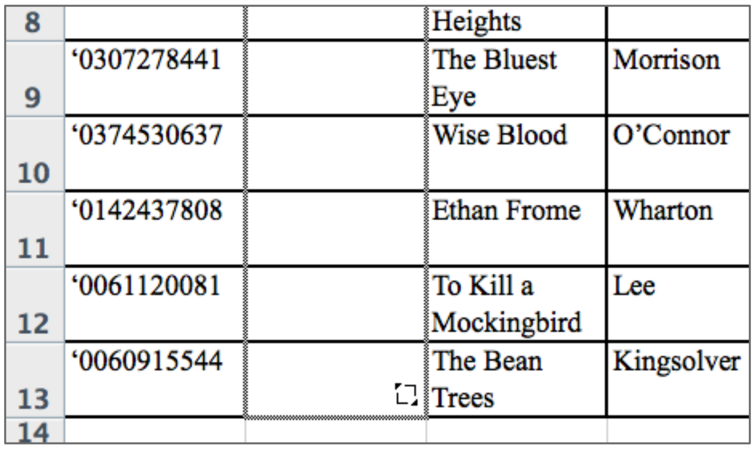
```
=RIGHT(A2,LEN(A2)-1)
```

d. In the second cell in column B, you should now see *the same number from column A,* but without the apostrophe.[3] (If that is not what you see, you made an error.)
e. Now you need to apply the same formula to all the cells in column B. This is a common task in Excel. **Hover** on the lower right corner of the cell B2 until you see *a plain black plus sign,* as shown below:



	A	B	C	D
1	ISBN		Title	AuthorLast
2	'0141439513	0141439513	Pride and Prejudice	Austen
3	'1844083721		Death Comes for the Archbishop	Cather
4	'0061120065		Their Eyes Were Watching God	Hurston
	'0679405828		Jane Eyre	Brontë

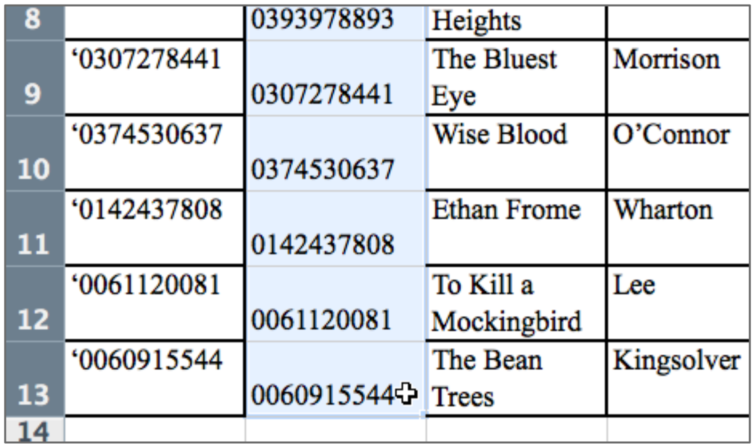
Then hold and *drag down* to include the last cell in column B:



	A	B	C	D
8			Heights	
9	'0307278441		The Bluest Eye	Morrison
10	'0374530637		Wise Blood	O'Connor
11	'0142437808		Ethan Frome	Wharton
12	'0061120081		To Kill a Mockingbird	Lee
13	'0060915544		The Bean Trees	Kingsolver
14				

[3] The formula you entered means: Starting from the RIGHT, take the contents in the cell A2, find the length of the contents (number of characters), and take the last one away (–1) *from the left side*. Show the result here in B2. LEN performs the length operation.

When you release the mouse, all the modified ISBNs will be in column B (without the apostrophes).



8		0393978893	Heights	
9	'0307278441	0307278441	The Bluest Eye	Morrison
10	'0374530637	0374530637	Wise Blood	O'Connor
11	'0142437808	0142437808	Ethan Frome	Wharton
12	'0061120081	0061120081	To Kill a Mockingbird	Lee
13	'0060915544	0060915544	The Bean Trees	Kingsolver
14				

f. Copy the **column heading** from cell A1 and paste it into cell B1.

9. **Drag to select** all the cells from B1 down to I13 (all the cells *except* the first column, where you have the apostrophe-ISBNs, which you *do not want* now).
10. **Copy.**
11. Open the Web page at:
12. http://www.shancarter.com/data_converter/
    Paste into the big box labeled "Input CSV or tab-delimited data."
13. **Copy** everything in the lower box, labeled "Output as [XML – Nodes]."
14. **Paste** the XML text into Dreamweaver, or into *a good plain-text editor*.[4]
15. Now we need to perform two find-and-replace operations to replace the generic "row" and "rows" written for us by the Mr. Data Converter application:
    a. In the XML file, **replace** all instances of "rows" with "booklist." Mr. Data Converter wrapped the whole list in the tag `<rows>`
    b. In the XML file, **replace** all instances of "row" with "book." Mr. Data Converter wrapped each data record in the tag `<row>` —that absolutely has to be `<book>` for our ActionScript to work in this exercise.
16. **Save** the document with filename *books.xml* (note that the file extension MUST be *.xml*). **This is your XML file.** The ActionScript in this exercise will look for this exact filename.

[4] Dreamweaver works great for this. If you use a decent text editor, such as Notepad++ (Win) or TextWrangler (Mac), these also do a good job. DO NOT USE standard MS Notepad (Win) or TextEdit (Mac) or MS Word for this. They will *mess up* the XML.

Check your XML file to see whether the *first* lines look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
      <bookList>
            <book>
                  <ISBN>0141439513</ISBN>
                  <Title>Pride and Prejudice</Title>
                  <AuthorLast>Austen</AuthorLast>
```

If not, *edit the file* to look like the lines shown above. (It's okay if the indents are smaller.)

The *final* lines in your XML document should look like this:

```
            <PubYear>1988</PubYear>
            <Genre>Novel</Genre>
            <Image>images/beantrees.jpg</Image>
      </book>
</bookList>
```

If you see unnecessary lines there (*above* the closing tag `</bookListData>`), delete them.

What did you just do?

Naturally this seems like a lot of work to do for 12 little data records. Please understand that normally you would be doing this for a *bigger* dataset. Think of U.S. Census data as an example. This is how you format data so that people can view (for example) 5,000 pop-up boxes on an interactive map.

You can use *this same procedure* to create an XML file of hundreds or thousands of data records.[5] Sometimes you might get an XML file that's already set up for you—but now you know how to make your own.

Examine any single record, and you'll see the eight data fields you started with:

```
<book>
  <ISBN>0199535574</ISBN >
  <Title>Sense and Sensibility</Title >
  <AuthorLast>Austen</AuthorLast >
  <AuthorFirst>Jane</AuthorFirst >
  <Publisher>Oxford</Publisher >
  <PubYear>1811</PubYear >
  <Genre>Romance</Genre >
  <Image>images/sense.jpg</Image >
</book>
```

[5] At a media organization, you would probably have an in-house converter application, so you would not need to use the one at http://www.shancarter.com/data_converter/

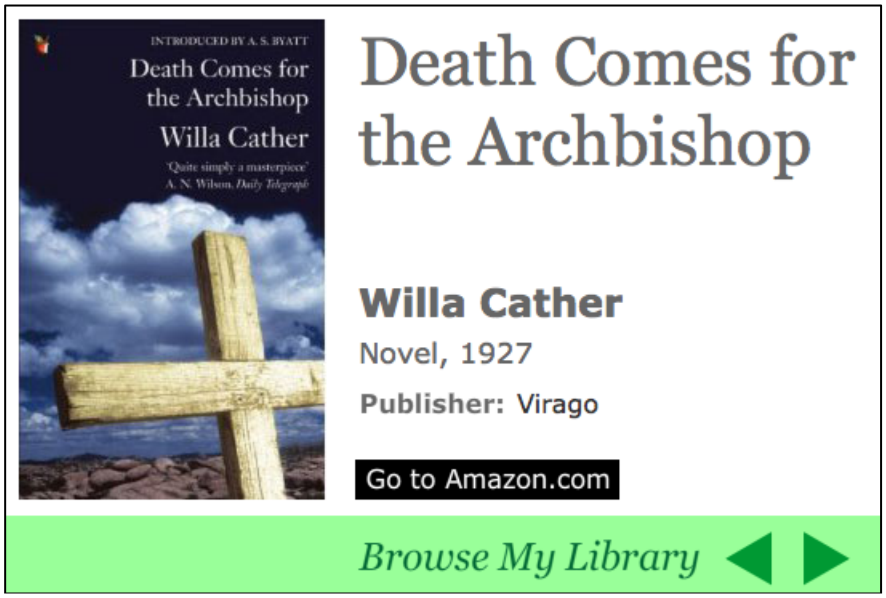
- Each record is enclosed by the same two tags: `<book>` and `</book>`
- The full dataset is enclosed by two tags: `<bookList>` and `</bookList>` — XML *requires* enclosing tags such as these.
- The XML tags work like HTML: *Opening* and *closing* tags are required for each data field and each data record.

Tags all have names that *you* create. They can be uppercase or lowercase, but they have to *match* (like HTML tags, like CSS). No spaces, and no punctuation. If you wanted to make XML for episodes of a TV show, the tags might look like this:

```
<episode>
  <season>2</season>
  <episodeNum>5</episodeNum>
  <origAirDate>10/15/2009</origAirDate>
  <title>Dream Logic</title>
  <summary>The Fringe team travels cross-country to Seattle after
learning of a mysterious incident involving a man who attacked his boss
because he believed he was an evil ram-horned creature. As these
puzzling occurrences continue, the team tirelessly explores strange and
creepy links to dreams. In pursuit of additional information, Agent
Broyles has a disconcerting meeting with enigmatic Massive Dynamic
executive Nina Sharp that leads the investigation in an unthinkable
direction.</summary>
</episode>
```

You may have *as many data fields as needed* in a data record. As you see above, a data field may contain a large amount of text.

Next: Using the XML in a multi-part package.


Death Comes for the Archbishop

Willa Cather

Novel, 1927

Publisher: Virago

Designing a screen layout with dynamic text fields

To create a screen or slide that will work for all the data and all the books, you can use the design above as a general guide (feel free to *change* the layout).

The widest book cover image is 206 pixels (width). All covers are 300 pixels high.

The buttons are provided for you in the *books.fla* file.

Create a layout in Flash using:

1. A plain rectangle shape with the dimensions of a book cover.
2. Four separate text boxes for title, author, genre and year, and publisher. Type and *style* text in the fields, and make them look good. Make the text in these four text fields Classic – Dynamic – Device Fonts[6] (**Properties** panel).
3. A text box that contains only the word *Publisher:* (This and any other text apart from the four dynamic fields should be Classic – Static – Anti-alias for readability. Make sure the text is NOT selectable.)
4. The Amazon button (provided).
5. Next and Previous buttons (one provided; use it *twice*—just flip it).

Naming text fields for data in Flash

Since I have already written the script for this, you will need to use the exact *instance names* used in that script. Those for the text fields are:

```
title_txt
author_txt
bookinfo_txt
publisher_txt
```

You have already created four dynamic text fields (step 2 on the previous page)—now give the appropriate *instance name* to each one, using the list above.

Next, give these instance names to the three buttons:

```
amazon_btn
next_btn
back_btn
```

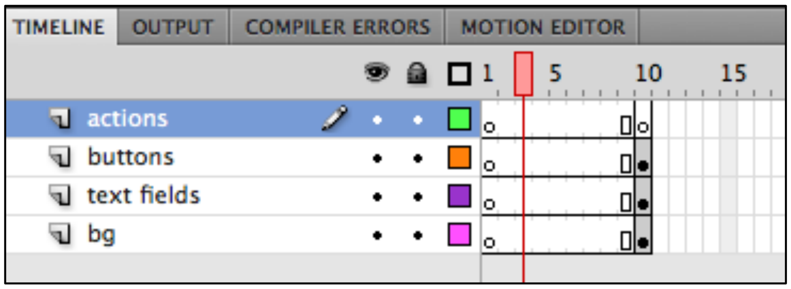
You are going to see all of these seven instance names in the ActionScript.

[6] Device Fonts are not bulletproof and should not be used for most Web applications. However, we want to avoid the font-embedding tasks today for this exercise, so this is a shortcut for class only.

Note that everything you did here on page 6 is simple Flash layout and design. Nothing special (except styling the fonts). Nothing difficult.

ActionScript 3 for loading XML in Flash

First you need to drag all your content into frame 10, leaving frames 1–9 empty as shown below:



Now you'll be putting ActionScript on frame 1 (in the "actions" layer, of course!). Here is the first bunch:

```
stop();

var myXML:XML;
var xmlLoader:URLLoader  = new URLLoader();
var itemNumber:uint = 0;
var myLoader:Loader = new Loader();

myLoader.x = 10;
myLoader.y = 10;
```

Explanation
The first four lines are variables. You are setting them up, or *initializing* them. Flash will use these later.

- **myXML** is for the XML data that will be loaded.
- **xmlLoader** is for the URL (or filename) of the XML file.
- **itemNumber** is for a data record number. When Flash reads your XML file, it will associate each data record with a unique number, starting with 0.
- **myLoader** is for an image loaded from an external file. Last week you used a Loader object for loaded SWF files. This is the same, only this time the filename for each image will come from the XML.

The next two lines position the loader, which is your image container. Get the X and Y coordinates from the rectangle you placed on the Stage in your layout and change them accordingly.

Now let's do some *work*. Write the next line:

```
xmlLoader.load(new URLRequest("books.xml"));
```

You are using one of you variables (**xmlLoader**) to get the XML file. That file must be in the same folder with your FLA and SWF. (*Do not* keep files on the Desktop!)

The following line is an EventListener that will "hear" when the whole XML file has finished loading. This is really important, because nothing works if this file does not load.

```
xmlLoader.addEventListener(Event.COMPLETE, handleMyData);
```

You can see that the listener is going to need a function (as usual!).

```
function handleMyData(e:Event):void {
     myXML = new XML(xmlLoader.data);
     play();
}
```

That function says:

- Get the variable **myXML** and stick into it *the data* (from the XML file) that came into the loader named **xmlLoader**.
- After that's done, play the timeline.

When the timeline plays, it's going to come to a stop on your final frame. Then everyone will see your lovely layout with the book cover and the text fields.

An important note

Everything you have done in the ActionScript up to this point is pretty much THE SAME every time you load any XML file for any reason. The *name* of the XML file (*books.xml*) will be different, no doubt. In future projects, you may or may not have images—or SWFs, or videos—specified in the XML file. If you *do,* you will need at least one loader. If you *don't,* then you will *not* need **myLoader,** and you will *not* need to set its X and Y.

You can use all the same variable names and function names you see here. I always use the same ones—that way, I just copy and paste all this code and the only thing I need to change is the name of the XML file.

Your code on frame 1 up to this point is:

```
stop();

var myXML:XML;
var xmlLoader:URLLoader  = new URLLoader();
var itemNumber:uint = 0;
var myLoader:Loader = new Loader();

myLoader.x = 10; // use your own X and Y numbers
myLoader.y = 10;

xmlLoader.load(new URLRequest("books.xml"));

xmlLoader.addEventListener(Event.COMPLETE, handleMyData);

function handleMyData(e:Event):void {
     myXML = new XML(xmlLoader.data);
     play();
}
```

Note that at this point, the XML data is already *in your SWF*. It is waiting to be used.

The function that does it all
It's nice to have one stand-alone function that puts all the XML data where you want it to appear. Then each time you want the data from a particular data record in the XML (in this case, the data for one book), you just run this function. It can run from a button or from a frame.

```
function makeSlide():void {
     myLoader.load(new
URLRequest(myXML.book[itemNumber].Image));
     addChild(myLoader);
     title_txt.text = myXML.book[itemNumber].Title;
     author_txt.text = myXML.book[itemNumber].AuthorFirst +
" " + myXML.book[itemNumber].AuthorLast;
     bookinfo_txt.text = myXML.book[itemNumber].Genre + ",
" + myXML.book[itemNumber].PubYear;
     publisher_txt.text = myXML.book[itemNumber].Publisher;
}
```

Those lines in blue are *not* new lines—they are part of the previous line, so DO NOT press return at the wrong place!

Now look at what each line does so you understand it:

```
myLoader.load(new
URLRequest(myXML.book[itemNumber].Image));
```

The image associated with this data record is going to go into **myLoader**. It is a *URLRequest* because the image is a file outside the SWF (just like the loaded SWFs last week). Let's also take a look at the meaning of `myXML.book[itemNumber].Image` —because this is a *pattern* you are going to see again in four lines just below this.

myXML refers to the whole loaded XML file.
book is the tag inside the XML file that wraps around one data record (the data for *one* of the books).
[**itemNumber**] is a variable you initiated back on page 7. It is still equal to 0. That would bring in the first book in your XML. If you wanted the sixth book, then **itemNumber** would need to be equal to 5. (You'll get more information about **itemNumber** below.)
Image, like **book**, comes staright out of your XML. This loader is going to get the filename that appears between the `<Image>` tags in the XML file. Note that the uppercase *I* is essential—it must match the XML file.

```
addChild(myLoader);
```

This line makes the loaded image visible. It is required. *Why* (you may ask) is the word "child" used? Let's keep it simple and simply say that in object-oriented programming, objects have relationships, and one way these relationships are configured is parent-child. *Objects on the Stage are considered "children" of something else.* If we do not include `addChild(myLoader)` when necessary, then the loaded content will never become visible.

```
title_txt.text = myXML.book[itemNumber].Title;
```

This is how text is written into your dynamic text fields. The information between the `<Title>` tags in the XML file is going to be written into your *title_txt* field on the Stage.

```
author_txt.text = myXML.book[itemNumber].AuthorFirst +
" " + myXML.book[itemNumber].AuthorLast;
```

The information between the `<AuthorFirst>` and `<AuthorLast>` tags in the XML file is going to be written into your *author_txt* field on the Stage. The part `+ " " +` adds a *space* between the two!

```
bookinfo_txt.text = myXML.book[itemNumber].Genre + ",
" + myXML.book[itemNumber].PubYear;
```

> The information between the `<Genre>` and `<PubYear>` tags in the XML file is going to be written into your *bookinfo_txt* field on the Stage.
> The part `+ ", " +` adds a *comma* and a *space* between the two.

```
publisher_txt.text = myXML.book[itemNumber].Publisher;
```

> The information between the `<Publisher>` tags in the XML file is going to be written into your *publisher_txt* field on the Stage.

The Next and Back button functions

In this example, we will just step through every record in the XML file. That means we want the Next button to *show next record, show next record, show next record* … until we reach the end of the list.

Likewise, we want the Back button to show the previous record, and the one previous to that, each time we click it.

Here is where **itemNumber** steps up to do its job. In computer code, every set of data objects is numbered, starting with 0. The second is 1, the third is 2, and so on. So to say, "Show me the next one," we tell the system to add 1 to the record number. To say, "Show me the previous one," we subtract 1.

And to make it nice and bulletproof, we have to test to see if we have reached the end yet (in either direction). Otherwise, Flash will throw an error.

```
function goNext(e:MouseEvent):void {
     itemNumber++; // add 1
     back_btn.visible = true;
     if (itemNumber == ( myXML.children().length() - 1 )) {
          next_btn.visible = false;
     }
     makeSlide();
}
```

Above is the function for the Next button. We'll go through it line by line.

```
itemNumber++;
```

> This is the same as saying: "Now make itemNumber *equal to* itemNumber plus 1." We are increasing the value of itemNumber by +1.

```
back_btn.visible = true;
```

We are going to ensure that the Back button is invisible when there are no previous records. However, since we just increased itemNumber by +1, then there must be a previous item, and so the Back button needs to be seen!

```
if (itemNumber == ( myXML.children().length() - 1 )) {
     next_btn.visible = false;
}
```

This is a very common kind programming statement called an *if-then*. The first line sets up a condition: *If this is true …* The second line provides a result or a consequence: *… then do this*. If the condition defined in the first line is not true, then the second line is ignored. The handy attribute `myXML.children().length()` tells us *how many records* are in the XML file. Then we have to –1 because (as you'll recall), the first one is numbered 0. So we are saying: If the itemNumber right now (after the +1 above) *is equivalent to* (==) the total number of records minus one, then make the Next button *not visible*. Get it? Because in that case, there will be *no next record*.

```
makeSlide();
```

This calls the function you wrote earlier—the one that does all the work of writing the text and showing the image.

The script for the Back button is *parallel to* the script above:

```
function goBack(e:MouseEvent):void {
     itemNumber--; // subtract 1
     next_btn.visible = true;
     if (itemNumber == 0) {
          back_btn.visible = false;
     }
     makeSlide();
}
```

```
itemNumber--;
```

This is the same as saying: "Now make itemNumber *equal to* itemNumber minus 1." We are decreasing the value of itemNumber by –1.

```
next_btn.visible = true;
```

We are going to ensure that the Next button is invisible when there are no more records left to view. However, since we just decreased itemNumber by –1, then there must be a next item, and so the Next button needs to be seen!

```
if (itemNumber == 0) {
     back_btn.visible = false;
}
```

Here's another *if-then* statement. The first line sets up a condition: *If this is true …* The second line provides a result or a consequence: *… then do this*. If the condition defined in the first line is not true, then the second line is ignored. Here we are saying: If the itemNumber right now (after the –1 above) *is equivalent to* (==) zero, then make the Back button *not visible*. There will be no record to go back to.

```
makeSlide();
```

This calls the function you wrote earlier—the one that does all the work of writing the text and showing the image.

Make sure you understand, now, what the role of itemNumber is.

Variables and a button for Amazon links

When I was making this tutorial, I wanted the XML data to include some kind of dynamic links to external Web pages. Since I copied the book covers from Amazon, I thought it only fair to use their pages for the links. Unfortunately they have a really long and unwieldy search URL. Buried in the middle of that URL is the book's ISBN—a universally used unique ID number.

So I did not need to include the complete URL in the XML. Most of it does not change—only the ISBN changes. So all that is needed in the XML is the ISBN.

However, here in the script, it's necessary to write out the complete URL. So I broke it into two parts: The part that precedes the ISBN, and the part that comes after. These two parts are written into two variables:

```
var amazon1:String =
"http://www.amazon.com/gp/search/ref=sr_adv_b/?search-
alias=stripbooks&unfiltered=1&field-keywords=&field-
author=&field-title=&field-isbn=";
```

```
var amazon2:String = "&field-publisher=&node=&field-
p_n_condition-type=&field-feature_browse-bin=&field-
binding_browse-bin=&field-subject=&field-language=&field-
dateop=&field-datemod=&field-
dateyear=&sort=relevanceexprank&Adv-Srch-Books-
Submit.x=15&Adv-Srch-Books-Submit.y=13";
```

These must be written with NO HARD RETURNS.

And then, a *third* variable is going to hold those two *plus* the ISBN:

```
var amazonURL:String;
```

What is *String*? It is a data type meaning the variable holds text (as opposed to a number or an integer).

All three of the variables come together with the ISBN (from the XML file) in this function, which will be called by the Amazon button:

```
function goAmazon(e:MouseEvent):void {
     amazonURL = amazon1 + myXML.book[itemNumber].ISBN +
amazon2;
     var myRequest:URLRequest = new URLRequest(amazonURL);
     navigateToURL(myRequest, "_blank");
}
```

The line in blue is *not* a new line—it is part of the previous line, so DO NOT press return at the wrong place!

Here's the walk-through for that function:

```
amazonURL = amazon1 + myXML.book[itemNumber].ISBN +
amazon2;
```

The value of the variable **amazonURL** will be: (1) the value of the variable **amazon1**, plus (2) the ISBN indicated by the current **itemNumber**, plus (3) the value of the variable **amazon2**. All three get strung together into one very, *very* long string!

```
var myRequest:URLRequest = new URLRequest(amazonURL);
```

Yet another variable declaration: This one is named **myRequest**. It is going to contain the current value of **amazonURL**—which was *just* created by the line above this one.

```
navigateToURL(myRequest, "_blank");
```

This is the normal way (in AS3) to open a new Web page in a new window. Because the variable **myRequest** holds the same giant string of text as **amazonURL**, that tells the browser which Web page to open.

We have come to the end of the frame 1 script! If you would like to copy and paste it, you can get it from the file *as3_to_add.txt.*

The script on frame 10
Add these lines to frame 10. This should all look quite familiar to you:

```
stop();

amazon_btn.addEventListener(MouseEvent.CLICK, goAmazon);
next_btn.addEventListener(MouseEvent.CLICK, goNext);
back_btn.addEventListener(MouseEvent.CLICK, goBack);

back_btn.visible = false;
makeSlide();
```

First, you have three button listeners. The buttons are not present until frame 10, so of course the listeners must be there.

You already wrote all three functions on frame 1. It is good to keep the functions on frame 1, even if they will not be used until later in the timeline.

goAmazon
goNext
goBack

Next, we tell the Back button to be not visible. Why is this necessary? Because at the top of frame 1, you set the value of **itemNumber** to 0. There is nothing to go back to (yet). Therefore, the Back button should not be clicked. If you don't want people to click a button, then make its visible attribute *false*—to hide it.

Finally, we call the function that does all the work: `makeSlide()`

This function will run each time Next or Back is clicked. However, you do *not* want people staring at a blank screen when they come to frame 10! That is why we run the function once from the frame itself—to populate the text fields and **myLoader**. ("Populate" is programmer's jargon.) If you want to see the difference, just comment out the line that calls the function by putting two slashes in front of it:

```
// makeSlide();
```

Save and test the movie. Everything should work beautifully (unless you have typos!).

Don't forget to go back and delete the two slashes you just added on frame 10. It works much more nicely that way.

The same XML data in a different design

Building on what you have just done in this exercise, you can examine the same XML data loaded into a different SWF, with the access to each book provided in a more user-friendly fashion.

View the second example here:

http://flashjournalism.com/CS4examples/XML/XMLexample.html

Then open the FLA (*example2.fla*) and check out the movie clip (in the Library) named Panel. The text fields should look familiar to you. This FLA is fully functional and fully coded.

What I would like you to recognize are two key similarities.

1. The way to load the XML file is the same (frame 1, Main Timeline):

   ```
   var myXML:XML;
   var xmlLoader:URLLoader  = new URLLoader();
   var itemNumber:uint = 0;

   xmlLoader.load(new URLRequest("books.xml"));

   xmlLoader.addEventListener(Event.COMPLETE,
   handleMyData);

   function handleMyData(e:Event):void {
        myXML = new XML(xmlLoader.data);
        play();
   }
   ```

   The ONLY thing you would need to change in a new project is the filename of the XML file.

2. The way to use the individual items in the XML file is the same. The fields and the loader are down inside a movie clip (Panel) which is animated inside another movie clip (Panel Animation), so everything needs to have two instance names appended on the front: *panelani_mc.panel_mc*. Strip those away and you see the same script as before (frame 1, Main Timeline):

```
function addContent(e:Event):void {
     panelani_mc.panel_mc.myLoader.load(new ¬
          URLRequest(myXML.book[itemNumber].Image));
     panelani_mc.panel_mc.addChild(panelani_mc. ¬
          panel_mc.myLoader);
     panelani_mc.panel_mc.title_txt.text = ¬
          myXML.book[itemNumber].Title;
     panelani_mc.panel_mc.author_txt.text = ¬
          myXML.book[itemNumber].AuthorFirst + " " + ¬
          myXML.book[itemNumber].AuthorLast;
     panelani_mc.panel_mc.bookinfo_txt.text = ¬
          myXML.book[itemNumber].Genre + ", " + ¬
          myXML.book[itemNumber].PubYear;
     panelani_mc.panel_mc.publisher_txt.text = ¬
          myXML.book[itemNumber].Publisher;
     amazonURL = amazon1 + myXML.book[itemNumber]. ¬
          ISBN + amazon2;
}
```

The symbol ¬ is standard in books about programming; it means that the line of script continues without a hard return. You must not try to copy and paste the code above. It will not work because of the ¬ symbol.

The big difference between these two examples is the way the individual data records from the XML are used. In the first example, the Next and Back buttons just stepped through the entire list. A user could not choose and could not skip around. In the second example, a grid of 12 buttons lets a user select any book, in any order.

That choice requires us to take some unique ID from the data record for each book and use it to associate the button with the data record. That is what enables us to view any record in any order. The ISBN is perfect for this because no two books in the world have the same ISBN.

The **switch statement** in the *openPanel()* function is very useful for a set of many buttons like this one. You can see how it frees you from writing 12 different functions. (However, you *do* still need 12 different listeners.)

The *openPanel()* function performs the following four tasks:

1. In the **switch** statement, **case** tests each possible case that might be true. Since 12 buttons use this function, there are 12 cases.

   `(e.currentTarget.name)` refers to whatever called this function (`e.currentTarget`) and gets its instance name. That means we get the button's instance name. Sweet!

   If a **case** is true, the value of the variable *isbn* becomes whichever one is written there under **case**, and then we quit from the **switch** statement (`break;`). If the **case** is not true, the script continues to the next case. At the end of this process, we have the ISBN for the book button that was clicked.

2. The **for loop** is very, very common in all programming languages. It allows us to loop through all the items in a list of some kind and do stuff to each one of them.

   ```
   for(var i=0; i< (myXML.children().length()); i++) {
        if (myXML.book[i].ISBN == isbn) {
             itemNumber = i;
             break;
        }
   }
   ```

   A **for loop** always starts with three items: a beginning point, a limit or ending point, and an increment (+1) or decrement (–1). In this example, the beginning point is 0 (1=0); the limit is the number of items (books) in our XML file (this comes from *myXML.children().length()* —that tells Flash how many items are in the whole list); the increment is accomplished by *i++* (you can look up more details about **for loops** online). A **for loop** typically uses *i* or *j* as its variable. With *var* inside the loop statement, the variable name cannot be used outside this function (it will not work).

   So what does this for loop **do** for each one of the items in our XML? It looks at each one (an item is enclosed by the tags `<book>` and `</book>` as I'm sure you will recall) and asks: "Does the ISBN number in this book data record *match* the value of the variable *isbn*, which we just got from the switch statement?" As soon as it *does* match, the **for loop** quits (`break;`) and we move on to the next thing. Oh, and the **value** of *i* is placed into the variable *itemNumber,* which is crucial to filling in our text fields (`itemNumber = i;`). That means Flash now knows *which* record we want from the XML. It is some number between 0 and 11—because *i* started at a value of 0 and ended at a value of *1 less than 12*.

Note: *myXML.children().length()* is equal to 12 for our XML.

3. The third step in the *openPanel()* function is to *call another function*:

   ```
   addContent(null);
   ```

   We discussed what that function does in detail on page 17 above.

4. The final step is—now that the content is all in place in the text fields and the loader, thanks to the *addContent()* function—play the movie clip Panel Animation. That will make the filled panel fade into view, just by playing it.

   ```
   panelani_mc.play();
   ```

The real-world version

It's actually a bit odd to manually create these buttons and place them on the Stage, as I have done for this example. I chose to do this because I hoped it would make it easier to understand how we make the jump from simply stepping through the XML list to actually using the data in a random way, choosing any item in any order.

In a real-world application, the buttons would probably be created via ActionScript, or at least the thumbnail image would be inserted into each button via ActionScript, and they would be positioned on the Stage or in a scrolling menu symbol by ActionScript as well. Thus everything would be automated and handled by ActionScript, and the XML file could be totally changed at any time without requiring ANY changes to the SWF or the FLA.

That's the beauty of external data.

You can imagine a weekly best-sellers list. Each week, the top 12 books are different. But all you have to change is the XML file. The SWF stays untouched on the Web server, and yet each week the SWF shows 12 different books and all their associated data.

The buttons would be populated from the XML file with two things: A thumbnail image, and the record number of the associated book data record. This would be accomplished with a **for loop**. It would say, in essence: "For each item, from item 0 to item 11, get the filename of the thumbnail. Place that thumbnail image into a loader inside the specified button symbol. Then place the current record number into a variable in that button symbol. Then (now that all the necessary information is inside the button), place an instance of that button on the Stage at a particular X and Y." When any button was clicked, we would not need to find the ISBN; we would already know the record number to use.