Building your own Flash MX Components
Phillip Kerman

Annotated presentation, downloads, and sample chapters from my two books (*Sams Teach Yourself Flash MX in 24 Hours* and *ActionScripting in Flash MX*) available at:
www.phillipkerman.com/devcon02/

__Overview
Components are not complex and elaborate code snippets that someone else built and installed in Flash MX (or uploaded to the Macromedia exchange site). Not necessarily. You'll see they don't have to be complex. They don't have to be elaborate. And, they're easy enough for anyone to build.

__The difference between a clip and a component

Every instance of a symbol on stage maintains its own properties. For example, you can set the position, scale, rotation, and color effects (like tint and alpha) differently for each instance you drag onto the stage. Using ActionScript you can refer to properties of individual instances during runtime using the form: object-dot-property—such as **clip1._rotation** where "clip1" is the instance name of a particular clip. To change the **_rotation** property of **clip1** you can simply assign it a new value as in **clip1._rotation=90**.

Similar to how clip instances maintain their own set of properties, custom variables become part of individual instances. You can refer to and change such variables using the *same* object-dot-property form. That is, **clip1.age=37** assigns the **age** variable in **clip1** to **37**. It's often easiest to simply think of variables in clips as a "homemade properties". Call them "variables" or "properties" (or even "parameters" if you want). The fact is, both built-in properties and custom variables are independent for each instance and use the same syntax. The fact that many custom variables have no visual representation onscreen is immaterial—you can always monitor variables with the Debugger or use the variables in other parts of your code. For example, when tracking the user's score you may not want to let them see the current value.

One difference between built-in properties and variables in clips is that built-in properties are set during authoring. Variables in clips are set during runtime with ActionScript. Setting variables is not difficult, but if you want the variables in each instance of the same symbol to start with different values you must set the variables from outside the clip. That is, initializing clip variables individually is achieved with a frame script *outside* the instance or in a **load** clip event script attached individually to each clip instead.

(Actually, it's possible to write a dynamic expression and place it inside the master symbol—perhaps in its first keyframe—that initializes variables uniquely based on a clip property such as **_name** but you'll see Components make this unnecessary.) The practical problem initializing variables outside the master symbol is that you either need to name each instance (so you can refer to the variable contained) or you need to place a separate (**load** clip event) on every instance. For every new instance you need to remember to either name it and set its variables or to attach a script to the individual instance. Therefore, your code becomes spread out and repeated in many places.

Components solve this issue by allowing you (the author) the ability to specify which variables (call them properties or parameters) can be set via the Parameters tab of the Properties panel by the author using the Component. In this way, you now have homemade properties that are changed via a panel.

__A Component is born

Converting a Movie Clip into a Component is easy. However, you should first determine which variables the using author needs access to. Then you simply use the Define Component dialog to specify which variables will be accessible to the using author. That's it. Components can be dragged onstage and, through the Parameters tab of the Properties panel, have their respective variables changed.

Examples of Simple Components

Style guide or template. Inside a Movie Clip, lay out a Dynamic Text field associated with the variable **title**. Use the Component Definition to allow the using author to specify the value for **title** (that is, the text that will appear). This Component can be used throughout a movie. The best part is that you can change the style or formatting of the text, and—since it's stored only once as a symbol—you'll see the changes reflected everywhere. Finally, a script placed in the first keyframe of the Component such as **_x=320; _y=240;** will make every instance appear in the same location regardless of where the sloppy author placed it.

Arrow button. Instead of having a separate button for "next" and another for "previous" (each with a slightly different version of the same basic code) place the button inside a Movie Clip. Turn it into a Component by allowing the using author to specify **direction** (either **1** or **-1**). Then the script you place on the button should use the value of **direction** to decide what script to execute. For example:

```
on (release) {
    _parent.gotoAndStop(_parent._currentFrame+direction);
}
```

Speaker Notes. The online version of this presentation loads variables containing about a paragraph of text for each slide. The user can rollover a button to reveal the appropriate text. I made a Component that let me (the using author) specify which variable would be associated with each instance I dragged on stage. I just dragged the Component into content areas (of my Flash movie) for which I had written speaker notes.

__Details of Component Definition

For every variable that you specify through the Component Definition dialog you may have both a "name" and a "variable". The name is for what the using author sees but the actual variable (that they're setting) can have another name. (Leaving "variable" blank means "name" gets used for both the using author and the actual variable name.) Additionally, you need to provide a default value (in case the using author never fills it in).

In addition, you can select from nine "types" which are comparable to data types. Basically, the type you choose affects how the using author will populate the variables. Here are the nine types:
Default is for individual numbers or strings. Using authors can basically set values to anything they want. If you want to force a **String** or **Number** or **Boolean** data type, use those selections.
List lets the author select from a popup menu. This way you limit the values from which the using author may select. In the end, your variables still contain just one string or one number value.
Array is quite different than **List**. Not only can the using author add and remove items (where in the **List** type is a predefined list) but in the end, the variable becomes an array data type with individual values in each index.
Object lets the using author set values for multiple properties of a single variable. The generic object data type (also known as "associative arrays") is like regular arrays with their multiple values, however objects have a name for each value. It's really like having a set of name/values within a single variable. This option is best when you need or want simple objects in your clip. It takes more effort to populate this type of variable but you can restrict the using author to a predefined list of properties like the **List** type. The two new cool features are **Font Name** and **Color**. **Font Name** provides a list of installed fonts and sets the corresponding variable to a string version of that font name. **Color** presents a standard Flash color picker and sets the variable to the integer version of that color. (Use **`someNumber.toString(16)`** to convert it to a hex string if you prefer.)

You can allow the using author to rename, add, or remove the parameters you defined by un-checking "Parameters are locked in instances". You can also provide a description that serves to explain how to use the Component.

Realize that the using author sets variables that are part of the main Component. You can use as many variables both contained in the Component and in nested clips—but through the Parameters tab of the Properties panel the using author simply sets variables of the Component. Only make the using author populate the variables necessary to make the Component behave uniquely.

__Custom User Interfaces (UIs)

You can make a separate .swf that plays inside the Parameters tab of the Properties panel (effectively replacing the name/value table). First, ask why. To simply make something more entertaining has questionable merit. There are two times that I see a need to create a Custom UI. First, if a better interface device can remove a tedious process of populating data—use it. For example, selecting colors by clicking a swatch or color selector is much easier than, say, typing the HEX values. The second reason to consider a Custom UI is when the using author needs to be guided through the populating process. Not so much to keep them from breaking things, but sometimes one variable's value will eliminate the need to gather another. In those cases it's nice to make a Custom UI that walks the user through the various settings like a "wizard". For example, the learning interactions Components that ship with Flash use tabs for this purpose.

Enough of the reasons *not* to make a Custom UI—here's how it's done. Identify the variables the using author will be populating. These must be stored in a generic object named **xch**. You can think of the **xch** instance as a surrogate for the actual Component—but just to hold the necessary variables as properties. You don't even need to instantiate this object like you'd expect (**xch=new Object()**). You can just go ahead and start setting properties of the **xch** instance as if it's been instantiated (**xch.myVar="value"**). Then you just build your UI movie to allow a user to set the key variables inside **_root.xch**. Building the script that lets the user interact is easy enough... even displaying highlights as they make selections is straightforward.

However, realize that unlike most movies that always initialize all variables the same way, the custom UI should visually reflect the values of variables set previously. That is, the using author may pull up the Parameters tab of the Properties panel to inspect clips they've already populated. The **xch** object will maintain those variables so they're initialized properly. However, you need to build some "restoration" script that re-displays highlights to represent the current values of the variables. Think of it this way: First, the using author clicks on a Component instance in the host movie. Then the Parameter tab in the Properties panel determines the variables of that instance and launches the Custom UI movie. An second later the Properties panel reinitializes variables in the Custom UI .swf's **xch** object. Then... the Custom UI is free to use the new values of variables contained in its **xch** object. Custom cursor example demonstrated is in my book *ActionScripting in Flash MX*—see workshop 2 at **www.phillipkerman.com/actionscripting**

__Additional Information

—Tip 1: Buttons can't be placed on top of buttons. While it would be very convenient to use the **on (rollOver)** mouse event that a button offers, if you plan to use your Component on top of buttons inside a movie you'll find the two buttons conflict. For example, if you make a custom cursor Component that lets the using author place it on top of any other button, you cannot use an invisible button inside the Component (to figure out when it's time to show the cursor). Instead use either the **getBounds()** or **hitTest()** methods.

Consider this code snippet:

```
onClipEvent (load) {
   rect=this.getBounds(_parent)
   _visible=false;
}
onClipEvent(mouseMove){
if(_xmouse>rect.xMin&&_xmouse<rect.xMax&&_ymouse>rect.yMin&&_ymouse<rect.yMax){
      active=true;
   } else {
      active=false;
   }
}
```

Using hitTest() may look cleaner, but notice that since you can't make the clip itself invisible you'll need to make its alpha 0:

```
onClipEvent (load) {
   //_visible = false;
   _alpha = 0;
}
onClipEvent (mouseMove) {
   if (hitTest(_root._xmouse, _root._ymouse, 1)) {
      active = true;
   } else {
      active = false;
   }
}
```

—Tip 2: If you plan to use **attachMovie()** to dynamically create clip instances inside your Component, be sure to include a copy of each symbol (being attached this way) in a guide layer. That way, when you copy the Component to another file, all the associated symbols are copied with it. Naturally, the guide layer isn't needed during runtime—it just assures all the symbols get copied.

—Tip 3: You can override built-in properties by including them when you set up the Component Definition. For example, specify that **_name** will be set through the Parameters tab. If you provide a default this makes it possible to automatically name every instance that's dragged on stage. By the way, this *will* override any settings you make through other panels such as the Properties panel.

—Tip 4: When you add parameters (through Define Clip Parameters), instances on stage may need to be replaced to reflect the new options.