

Creating ActionScript 3.0 components in Flash CS3 Professional – Part 3: From prototype to component

Jeff Kameron

Adobe

Welcome to the third part of this article series on creating components with ActionScript 3.0. If you skipped the first two installments, you might want to begin with Part 1 of the series where you can download the sample files for the entire series. Or if you prefer, you can download the sample files for Part 3 only to follow along with this article.

As we pick up from Part 2 of this article series, we will take the final prototype we created—which is close to being structured in a basic component form, and we'll go through the process I used to turn it into an ActionScript 3.0 component. As I made my way towards this goal, I had to make some important changes to the project. Looking back to the steps we covered in Part 2, you'll remember that I took a big shortcut to get the prototype started. Rather than creating the menu prototype using ActionScript, I dragged TileList and List component instances out to the Stage and customized them by setting their parameters in the Component inspector.

At this point, the prototype has a fixed number of drop-down menus, a fixed height and width for the menu bar, and a fixed set of labels for both the menu bar items and the drop-down menu items. The whole idea behind a MenuBar component is that it should expose all these things for the user to control via parameters, so all of these properties need to be set up dynamically. In order to make this happen, it was necessary to restructure my FLA file and also alter my code.

You'll see how this all comes together as we step through the concepts presented in this article. So, let's get started. First, we'll take a look at the structure of components, to get a better understanding of how to modify the prototype to fit that convention.

Examining the basic two-frame structure of components

Component movie clips have two frames. The first frame always contains only one instance on stage, an avatar that is only there to define the default dimensions of the component. The second frame contains the assets that will be available to the component ActionScript code at runtime and are created dynamically. This two-frame approach is very similar to the structure typical of ActionScript 2.0 components, with the exception that with ActionScript 3.0 components, there isn't a `stop()` method on Frame 1. In fact, frame scripts cannot be used at all since components inherit from `Sprite` but not `MovieClip`. See the sidebar [Sprite vs. MovieClip](#) for more explanation on this subject.

The two frame tango

One nuisance caused by the two-frame structure of components, which anyone who uses ActionScript 3.0 components will invariably notice, is that when a FLA file using components has any compile errors, the components will jump back and forth between these two frames in the Flash Player. This occurs in situations when there are any compile errors at all, because the ActionScript 3.0 byte code isn't put into the exported SWF file, which means that the component movie clip is not linked to a class inheriting from `Sprite` or `MovieClip`, but rather is seen by Flash Player as just a normal movie clip. As a result, it follows default movie clip behavior and plays through its frames, endlessly looping. This behavior is seen not just when errors are present in the component code itself, but will occur when any error at all is present in the code, so all Flash developers will see it eventually. This happens with both FLA-based and SWC-based components, assuming that they use the two-frame architecture.

The prototype's MenuBar symbol was a one-frame movie clip with all the necessary component instances placed on stage, with instance names and customized components. While a simple component could be set up this way, it is very limiting since it makes the component fairly static, with a specific number of instances that have specific sizes. Most components lay themselves out quite differently depending on the dimensions and settings of the parameters.

After making my initial changes to the MenuBar symbol's Timeline, it was updated to match this two-frame structure (see Figure 1).

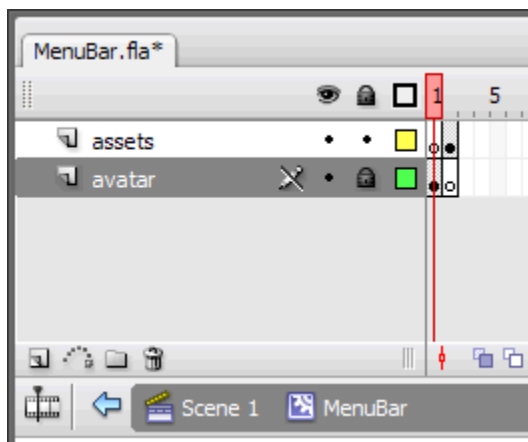


Figure 1. A second frame was added to the Timeline to match the basic component structure

Avatar layer

The avatar layer contains a rectangle with a hairline stroke and no fill, placed at (0, 0). The rectangle has width of 400 pixels and height of 22 pixels, which defines the default dimensions of the component. Choosing the default dimensions is pretty arbitrary and I

chose 400 x 22 simply because that was the dimensions I was working with in the prototype.

It is very important that this rectangle have a hairline stroke! The bounds of a shape with a wider stroke are not always the same bounds that you see in the Property inspector, and this discrepancy could cause the dimensions of a component placed on Stage to be different than expected at runtime.

The avatar can be a simple shape or it can be a graphic symbol or a movie clip symbol, or anything really. All that is important is that it is the only object on the Stage in Frame 1. At runtime it will be removed from the Stage and discarded. As described earlier in the sidebar Understanding the option: Automatically declare stage instances, it is important that this instance **not** have an instance name.

This layer is locked, since it should never be necessary for a user of the component to edit it. The layer is not hidden. You should never hide layers in your component movie clip because the option to Export hidden layers in the Publish settings could be unchecked in the user's FLA file.

Assets layer

There are two important requirements for every Library symbol that will be dynamically instantiated by a component:

- The symbol must be on Frame 2 in the component symbol
- The symbol must have the Export for ActionScript option checked in the Linkage dialog box
- The symbol must **not** have Export on first frame checked in the Linkage dialog box

The assets layer contains all symbols that will be required at runtime for the component. In the case of our very simple component, the only two symbol assets required are the List component and the TileList component, so one instance of each is put on the Stage. As was discussed earlier in the sidebar Understanding the option: Automatically declare stage instances, it is important that the component instances **not** have instance names and that all of the parameters for the component instances are left as the defaults.

When the user drags a component from the Components panel, Flash imports the component symbol into the Library and also imports every symbol on Stage within that component into the Library. This is exactly the same behavior you see when you copy and paste a symbol from one FLA file to another. So if any symbol assets were *not* on Frame 2, they would not be imported into the Library. Also, since Export on first frame is unchecked for the asset symbols, they will not be exported to the SWF unless they are on the Stage somewhere, so putting the symbols on Frame 2 ensures that they will be exported to the SWF along with the component symbol.

The assets layer is left unlocked. This location is commonly used to place the user-editable symbols, so by convention is not locked. This component symbol is not set up for user editing yet, but in the Advanced FLA Structure section, I discuss how to add editable skins to the assets layer. The assets layer is not hidden. As mentioned before, you should never hide layers in your component movie clip because the Export hidden layers option in the Publish settings could be unchecked in the user's FLA file.

Why is it important to uncheck Export on first frame for asset symbols?

By not checking the Export on first frame option, we make it easier for the user to toggle this checkbox for the component and control how the component's assets are exported to their SWF file. Selecting this Export on first frame for a symbol ensures its definition is exported into the published SWF file, whether or not it is on the Stage. Since all of a component's assets are on the Stage in Frame 2 of the component symbol, they will be exported along with the component symbol, effectively sharing its Export on first frame setting.

For example, if a user had the Button component in his Library, but did not drag it on the Stage and they unchecked the Export on first frame option, then the Button component would not be exported to the SWF at all, and neither would any of its assets—since none of them have the Export on first frame option checked either. On the other hand, if those asset symbols had Export on first frame checked, then they would all be exported to the SWF, bloating its size, even though the Button component is not exported, and the user would need to go through his Library and uncheck Export on first frame for every asset symbol. The Button component, which is a simple component, has 12 asset symbols. The DataGrid component has 42 asset symbols. You can imagine how difficult it would be for a user to manage the properties of all these assets separately.

By using our approach, the user can simply toggle the checkbox for the component itself to control whether any of the dependent assets will be automatically exported on the first frame. If we checked the Export on first frame option for every dependent asset, the user would need to change the properties for many different Library symbols to control where the assets were exported, which could be a headache.

Extending `fl.core.UIComponent`

To use the User Interface Component Infrastructure, your component class must inherit from `fl.core.UIComponent`. In the case of the `MenuBar` example we are working with, I extended `UIComponent` directly, using the code below:

```
import fl.core.UIComponent;
public class MenuBar extends UIComponent {
```

However, a class linked to a component movie clip symbol does not need to extend `UIComponent` directly. For example, `fl.controls.Button` extends `fl.controls.LabelButton`, which extends `fl.controls.BaseButton`, which extends `fl.core.UIComponent`. Depending on the type of component you are developing, it might

work best to extend `UIComponent`, or you may want to extend a component class, like `CheckBox` or `ProgressBar`, or you might want to extend one of the shared base classes used by the components, like `SelectableList`, which is the base class for `DataGrid`, `List` and `TileList`.

Another change I made to the source was to add variable declarations for the instances that had been on the Stage. In the prototype, these variable declarations were automatically added by Flash for me. As was discussed earlier in the sidebar [Understanding the option: Automatically declare stage instances](#), it is important that component movie clips do not have any named instances on the Stage. Additionally, you should always declare all variables in a component class and never have any variables automatically declared for you.

```
// a TileList for the menu bar
protected var myMenuBar:TileList;

// list of Lists for drop-down menus
protected var myMenus:Array;
```

The initial version of the component class has many changes from the prototype in order to support the change from `menu1`, `menu2`, `menu3` and `menu4` to the `myMenus` array. I do not cover these changes in detail in this section, but the ActionScript code is heavily commented. I recommend that you download the source files if you haven't already, and read through the comments and step through the ActionScript using the debugger to understand the code thoroughly.

Not extending UIComponent

It is possible to create a component without extending `UIComponent` or one of its subclasses. Such a component could not leverage the User Interface Component Infrastructure at all, so it would not be able to take advantage of the styling and skinning support, the invalidation model or the focus management, among other things. The Infrastructure does many things for you, so if you choose to create a component that does not use the Infrastructure, you'll need to write all of the code to do these things yourself.

If the component you are building diverges sharply from the Infrastructure, then it makes sense to write your own code to do these things. The `FLVPlayback` component is not based on the Infrastructure for this reason, as well as other historical reasons. The `FLVPlayback` component has a very different skinning model, it has a simple enough user interface that the invalidation model is not necessary and it does not support keyboard focus. The `FLVPlayback` component includes code that allows it to do some of the things that the Infrastructure does. For example, it has code to set the `isLivePreview` property, to remove the avatar shape and it also overrides the `width`, `height`, `scaleX` and `scaleY` properties to manage its own dimensions.

It is unlikely that you will find a compelling reason to create components without using the User Interface Component Infrastructure, and you will find it more difficult as well. But it is important to point out that it is possible to create a component without extending `UIComponent`.

Implementing a `configUI()` method

The method `configUI()` is called by the `fl.core.UIComponent` constructor after the important initialization of styles and invalidation support is completed. Put your initialization code that involves creating display objects, initializing them and adding them to the display list here. The method `configUI()` is a better place for this initialization than the constructor because your display object creation can interact with styles or invalidation processes, especially if the display objects created are subcomponents, as they are in our `MenuBar` component example.

An overriding implementation of `configUI()` should always call `super.configUI()` first. The `UIComponent` implementation of `configUI()` does a lot of important initialization, including initializing the `width` and `height` properties so that they can be used by your `configUI()` code. They also do the job of removing the avatar shape from the display list, which should always be performed before the code in your component starts adding items to the display list.

Let's look at the constructor and the `configUI()` method for `MenuBar` to see an illustration of where to place each section of initialization code:

```
public function MenuBar() {
    // initialize to an empty Array to avoid annoying null checks
    myMenus = new Array();

    // initialize dataProvider to non-null to save us from null checks later
    if (dataProvider == null) {
        dataProvider = new DataProvider();
    }
}

override protected function configUI():void {
    // always call super.configUI() in your implementation
    super.configUI();

    // dynamically create myMenuBar
    myMenuBar = new TileList();
    addChild(myMenuBar);

    // configure myMenuBar
    myMenuBar.selectable = false;
    myMenuBar.setStyle("cellRenderer", CellRenderer);
    myMenuBar.addEventListener(MouseEvent.CLICK, menuBarMouseHandler);
}
```

In the constructor, I initialized the properties `myMenus` and `dataProvider`, neither of which are display objects. Even if you are not familiar with a `DataProvider`, it should be obvious that it is not a display object because I did not call `addChild()` or `addChildAt()`—it would be unusual to create a display object and not add it to the display list. In `configUI()`, I created and initialized `myMenuBar`, which is a display object added to the display list. Notice that I did not add the code to create `List` instances for the drop-down menus here; I didn't add it because there is always one `TileList` for the menu bar, but there can be different numbers of `List` instances, depending on the contents of the menu. So, it is important to note that not all of your display objects should be created and customized in `configUI()`, but typically you would add any display objects here that will be created once upon initialization and then kept around while the `MenuBar` component is being used.

Example of replacing display objects in `configUI()`

`MenuList` is a class, which will be added later when style support is added, but it has an interesting `configUI()` method, so let's jump ahead and look at it now.

```
package fl.example.menuBarClasses {

    import fl.controls.List;
    import fl.controls.ScrollPolicy;

    public class MenuList extends List {

        override protected function configUI():void {
            super.configUI();

            // remove _verticalScrollBar and replace with a NoScrollBar
            removeChild(_verticalScrollBar);
            _verticalScrollBar = new NoScrollBar();

            // remove _horizontalScrollBar and replace with a NoScrollBar
            removeChild(_horizontalScrollBar);
            _horizontalScrollBar = new NoScrollBar();

            // to be safe, set both scroll policies to OFF
            _horizontalScrollPolicy = ScrollPolicy.OFF;
            _verticalScrollPolicy = ScrollPolicy.OFF;
        }

    }

}
```

The reason I created a `MenuList` subclass of `List` is that our `MenuBar` component will never need a scrollbar, but in the `configUI()` implementation of `fl.containers.BaseScrollPane` (a superclass of both `List` and `TileList`), `List` creates two `ScrollBar` instances. Since the `MenuBar` component will never need scrollbars, I deleted the `ScrollBar` skin assets from the FLA file, which meant that the two `ScrollBar` instances were throwing runtime errors when their `draw()` methods were called. By

removing them immediately, I was able to keep them from getting invalidated, so they were never drawn.

However, leaving the properties `_horizontalScrollBar` and `_verticalScrollBar` equal to `null` also caused runtime errors, so I replaced them with my benign version, `NoScrollBar`, which I never bothered adding to the display list since it will always be invisible. Earlier, I said that not adding a display object to the display list is unusual, and it is, but this is just one of those unusual moments. `NoScrollBar` is a very simple subclass of `ScrollBar` that is very careful never to draw itself:

```
package fl.example.menuBarClasses {

    import fl.controls.ScrollBar;
    import fl.core.InvalidationType;

    public class NoScrollBar extends ScrollBar {
        /*
         * override invalidate() so it does nothing. This should keep draw()
from being called
         */
        override public function
invalidate(property:String=InvalidationType.ALL,callLater:Boolean=true):void
{
    }

        /**
         * override draw() so it does nothing, just in case it is actually
called.
         */
        override protected function draw():void {
        }

    }
}
```

If this example seems confusing, don't fret. It is pretty complicated, so you might want to glean what you can now, then come back and look at this section again later after you have read the entire series. It took me many attempts and a bit of trial and error to stumble upon this solution as the simplest hack around my scrollbar problem, and as always, searching through the component code and stepping through the code examples in the debugger were invaluable to me. I intentionally made the `MenuBar` component example somewhat complex in the hope of illuminating some of the tricks and hacks useful in creating components—at the risk of leaving some readers scratching their heads. I don't know about you, but I get frustrated with articles that have trivial examples.

In the next section of this article, we'll take a look at how to work with the `draw()` method and the dynamic assets that display for the `MenuBar` component.

Implementing a draw() method

While `configUI()` is the place for initialization when a component is created, `draw()` is the place to put code that will be called every time changes to a component require it to draw itself. When I say draw, this could refer to using the drawing API via `flash.display.Graphics`, but more often it means dynamically creating instances of display objects and adding them to the display list, making instances visible or invisible, changing instances' dimensions and moving instances within the component. In the case of the `MenuBar` component, `draw()` is where the `List` instances for the drop-down menus are created and configured. It is also the place where the `dataProvider` for `myMenuBar` is configured.

A call to `draw()` is triggered by a prior call to `invalidate()`, which will be covered more in depth when I discuss invalidation in Part 6 of this article series. Your code should never call the `draw()` method directly, although you may sometimes need to call `drawNow()`, as explained in the following section. In the `MenuBar` example, I call `invalidate()` from the setter for `dataProvider` and from the `handleDataChange()`, the event handler called when changes are made to the `dataProvider`. For other situations that require invalidation, such as dimensions being changed via changes to properties like `width` or `scaleY`, style changes and changes to the property `enabled`, the User Interface Component Infrastructure handles the necessary invalidation.

As the last line in your `draw()` method, you should call `super.draw()`. The `UIComponent` implementation of `draw()` does two important things for you: It draws the `focusRectSkin` for you, when necessary, and it calls `validate()`, which clears the invalidation state.

Most of my initial implementation of `draw()` uses simple math to lay out the subcomponents. Check out the code below:

```
override protected function draw():void {
    // first clear out everything.
    clearMenus();

    // resize the menu bar
    myMenuBar.width = width;
    myMenuBar.height = height;
    myMenuBar.rowHeight = height;

    // fill the menu bar and create the drop-down menus for each dataProvider
    entry
    for (var i:int = 0; i < _dataProvider.length; i++) {
        initMenu(_dataProvider.getItemAt(i));
    }

    // if we have menus, then we set up the rowHeights, heights, widths and
    locations of everything
    if (myMenus.length > 0) {
        // distribute the menus evenly across the menu bar
        myMenuBar.columnWidth = (myMenuBar.width / myMenus.length);
        // make each drop-down menu below the corresponding menu bar cell,
```

```

        // make it the same width as the cell and match its height to
        // its contents
        for (var j:int = 0; j < myMenus.length; j++) {
            var theMenu:List = (myMenus[j] as List);
            theMenu.x = (myMenuBar.columnWidth * j);
            theMenu.y = myMenuBar.height;
            theMenu.width = myMenuBar.columnWidth;
            theMenu.height = theMenu.dataProvider.length * theMenu.rowHeight;
            // This line is very important! Invalidating a subcomponent
during a validate() or
            // call to draw() can leave things in a funky state where the
subcomponent is never
            // validated. When things are not updating as they should, try
calling drawNow()!
            theMenu.drawNow();
        }
    }
    // see my previous comment on the use of drawNow()
myMenuBar.drawNow();

    // always call super.draw() at the end
super.draw();
}

/*
 * removes all event listeners, wipes the data from the menu bar
 * and destroys all drop-down Lists from myMenus array
 */
protected function clearMenus():void {
    closeMenuBar();
    myMenuBar.dataProvider = new DataProvider();
    while (myMenus.length > 0) {
        var theMenu:List = (myMenus.shift() as List);
        removeChild(theMenu);
    }
}

/*
 * sets up the menu item for given index. Grabs the dataProvider
 * element for that instance, uses label for the item label in
 * the menu bar, creates a drop-down menu and parses the comma
 * delimited data to populate the drop-down menu.
 */
protected function initMenu(item:Object):void {
    myMenuBar.dataProvider.addItem({label:item.label});
    var str:String = item.data;
    var tokens:Array = str.split(",");
    var theMenu:List = createMenu();
    for (var i:int = 0; i < tokens.length; i++) {
        theMenu.dataProvider.addItem({label:tokens[i]});
    }
}

/*
 * creates drop-down List instance.
 */
protected function createMenu():List {

```

```

    var theMenu:List = new List();
    theMenu.visible = false;
    theMenu.selectable = false;
    myMenus.push(theMenu);
    addChildAt(theMenu, 0);
    return theMenu;
}

```

An `fl.data.DataProvider` instance determines the contents of the menu bar and the drop-down menus. The `dataProvider` property for `MenuBar` holds the same data as the `dataProvider` for `List`; both are lists of objects with `label` and `data` properties. For the `MenuBar` component, the `label` identifies the menu bar item label and the `data` is a comma-delimited list of drop-down menu items. The code that handles each object from the `DataProvider` is in `initMenu`. I discuss the `dataProvider` property more closely in the ActionScript metadata section of this article.

Every time `draw()` is called, it destroys and recreates all of the drop-down menus. Later, in the Invalidation section, I will leverage the invalidation model to be more elegant about what needs to be changed in `draw()`.

Understanding the importance of `drawNow()` and `validateNow()`

As you read through the `draw()` method, you will notice a comment that points out the calls of the method `drawNow()`. When working with components, the functions `drawNow()` and `validateNow()` are never to be forgotten, because sometimes they are the only things that will solve problems that would otherwise leave you scratching your head.

If your subcomponents are not updating correctly and the code looks right, my advice is to start adding calls to `drawNow()`, and if that does not work, try `validateNow()`. These methods function as safety valves if components are not rendering properly. You should always try `drawNow()` before `validateNow()` because in some cases `validateNow()` will cause more redrawing than is necessary and therefore can be less efficient.

`drawNow()` and `validateNow()` under the hood

A call to `drawNow()` forces an immediate call to `draw()`. A call to `validateNow()` forces a complete invalidation and then immediate call to `draw()`. The code for these methods, both defined in `UIComponent`, is pretty simple so let's take a look at it:

```

public function validateNow():void {
    invalidate(InvalidationType.ALL, false);
    draw();
}

public function drawNow():void {
    draw();
}

```

The implementation of `drawNow()` simply calls `draw()`. This may seem redundant, but it is necessary because `draw()` is protected. In the `draw()` example above from `MenuBar`, I could not have called `draw()` on the subcomponent instances, but I could call `drawNow()`, because it is public.

The method `invalidateNow()` passes `InvalidationType.ALL` as the first parameter to `invalidate()`, ensuring that every code path in `draw()` that checks for any type of invalidation will be hit. Next, it passes `false` as the second parameter, preventing the scheduling of a `draw()` call on the next `enterFrame` event. Then it calls `draw()`.

In the case of the `draw()` implementation of `MenuBar`, the problem is that an invalidation during a call to `draw()` often fails to schedule a call to `draw()` on the next `enterFrame` event. This is by design—if the `enterFrame` event handler calls `draw()` after an invalidation, it is not allowed to schedule another `draw()` on the next `enterFrame`. This restriction prevents a component from invalidating itself in its `draw()` method, causing `draw()` to be called on every `enterFrame` event forever.

Exposing component parameters and ActionScript metadata

You'll find ActionScript metadata throughout all component code: in ActionScript 2.0 and ActionScript 3.0, in Flash and in Flex. ActionScript metadata comes between square brackets. It has a name that is followed by a list of name/value pairs enclosed in parentheses. For example, here is metadata with the name `Inspectable` and a single name/value pair, `defaultValue/Label`:

```
[Inspectable(defaultValue="Label")]
```

If there is only one value, the metadata can actually be nameless, in which case it looks like this:

```
[IconFile("FLVPlayback.png")]
```

When there are multiple name/value pairs, it looks like this:

```
[Inspectable(defaultValue="vertical", type="list", enumeration="vertical, horizontal")]
```

ActionScript metadata applies to the definition that follows it. With Flash components, the metadata always applies to either a class definition or a property definition. Note that the same metadata is not supported for Flash components as is for Flex components, and sometimes if the same metadata is supported it does not mean exactly the same thing to Flash as it does to Flex. Please refer to Flex 2 documentation for more information on the metadata it supports.

Property level metadata should appear once, before the definition of either the setter or the getter for the property; do **not** put duplicates of the metadata before each function.

When overriding a property with metadata attached to it, you'll often need to duplicate the metadata from the base class, attaching it to the implementation in your subclass. This is not always the case. For example, if the metadata in the base class is before the setter function and you only override the getter function, you will not need to duplicate the metadata. It is always safe to duplicate the metadata from the base class when overriding a property, so you are free to always do it and not keep track of whether it is necessary in a specific case.

Inspectable metadata

Inspectable metadata is the most common metadata. It acts on properties and exposes them as component parameters in the Component inspector and Property inspector. Inspectable metadata supports the properties type, defaultValue, enumeration, verbose and name.

Type

The property type determines the type of the component parameter. It is not required and if it is not specified, then it will be determined by the ActionScript type of the property. The following are supported values for type:

Parameter Type	ActionScript Type Information	Default Value
String	default for <code>String</code>	Empty string
Number	default for <code>Number</code> , <code>int</code> and <code>uint</code>	0
Boolean	default for <code>Boolean</code>	false
Array	default for <code>Array</code>	null
Color	property should be of type <code>uint</code>	#000000
List	property should be of type <code>String</code>	First element in the enumeration

There are also types that are specific to the FLVPlayack component which are not documented here.

defaultValue

The property defaultValue specifies the initial value the parameter will have when an instance of your component is dragged onto the Stage. The property defaultValue is not required, and it will default to the values listed in the table above if a default is not specified. The default value for a parameter of type Array should be specified as a comma-delimited list. The default value for a Color should be in the form #RRGGBB. The default value for a List should be one of the values listed in the enumeration property for the parameter.

Note that all of these default values are only for the parameters set through the Component inspector and the Property inspector. When a component is created dynamically using ActionScript, these defaults will not apply. As the component developer, it is up to you to make sure that the defaults specified for the component parameters match the defaults initialized by the constructor.

enumeration

The property enumeration is only used with parameters of type List; if the type is not specified and the enumeration property is used, then the type is assumed to be List. The enumeration property is a comma-delimited list of possible values for the parameter. The user will be able to choose one of these values from a drop-down list in the Component inspector. Here is a code snippet example from BaseScrollPane.as, with the comments removed:

```
[Inspectable(defaultValue="auto",enumeration="on,off,auto")]
public function get horizontalScrollPolicy():String {
    return _horizontalScrollPolicy;
}
public function set horizontalScrollPolicy(value:String):void {
    _horizontalScrollPolicy = value;
    invalidate(InvalidationType.SIZE);
}
```

verbose

If the property verbose is set to true, then the property will only appear in the Component inspector and will not appear in the Property inspector. The verbose property is optional and it defaults to a value of false.

name

The property name determines the component parameter name that is visible in the Component inspector. The property is optional, and if it is not specified then the name will match the name of the class property. It can be confusing to users when a parameter name does not match the corresponding property in the class, so using this property is not considered a best practice.

Collection metadata

Collection metadata also exposes a property as a parameter. A parameter defined by Collection metadata will raise the Values dialog box, which allows the user to specify a list of objects that fit a specific pattern. Collection supports the properties collectionClass, collectionItem, identifier and name.

For the MenuBar component, I supported a very simple dataProvider property that, like the dataProvider property for List, takes a label field and a data field. The label field

defines the label for the menu bar item and the data field provides a comma-delimited list of items for the corresponding drop-down menu (see Figure 2).

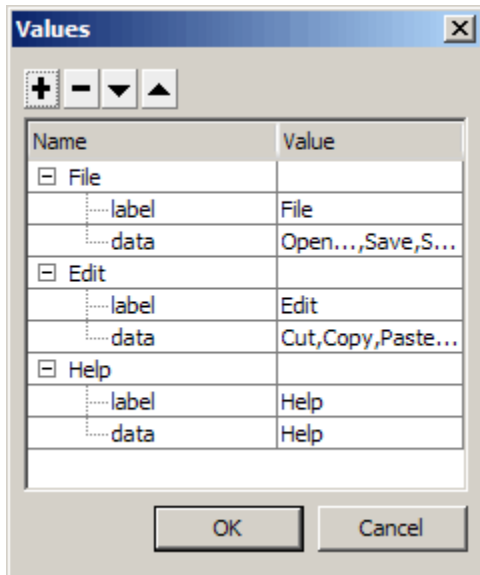


Figure 2. The Values dialog box displays the label and data values

I did this so I could base my `dataProvider` implementation, including the `Collection` metadata, on the implementation found in `SelectableList.as` and thereby save time and effort. If you compare my code to the code in `SelectableList.as`, you will see that my `dataProvider` getter and setter function and my `DataEvent` handling code is a simplified version of that code.

collectionClass

The `collectionClass` property determines the class that will be created and used to set your property. It is required. The type of your property does not need to match the `collectionClass` property precisely, but the type should be a superclass of the `collectionClass` property or be an interface that it implements or you will encounter errors. The requirements for the class specified by the `collectionClass` property are:

- It must have a constructor that takes no arguments.
- It must have a method `addItem()`, which can take objects of the type specified in `collectionItem` property. The return value does not matter.
- The class does not have to implement any specific interface or extend any specific base class.
- Specifying the class in the metadata does not necessarily force its compilation into the SWF file. This class must be referenced somewhere else in your code to guarantee inclusion.

collectionItem

The `collectionItem` property determines the class created for each item in the collection. It is required. The list of inspectable parameters for this class determines the list of properties shown in the Values dialog box for each item. Therefore you cannot simply specify `Object`, you must specify a custom class with Inspectable metadata. The requirements for the class specified by the `collectionItem` property are:

- It must have a constructor that takes no arguments.
- It must have at least one inspectable parameter.
- The inspectable parameters should be of type `String`, `Number` or `Boolean`. The more complex types, like `Color` and `Array`, are not supported.
- The class does not have to implement any specific interface or extend any specific base class.
- Specifying the class in the metadata does not necessarily force its compilation into the SWF file. This class must be referenced somewhere else in your code to guarantee inclusion.

identifier

The `identifier` property specifies which of the inspectable parameters in the `collectionItem` class is used to label the collapsible line for each item in the Values dialog box. This parameter will also be auto-populated with "`<property name><increasing ordinal>`" when an item is added in the Values dialog box, so it needs to be of type `String`. For example, the `dataProvider` parameter for `MenuBar` specifies `label` as the identifier. As you can see in Figure 2 above, the label value is used in the collapsible line. When more items are added in the Values dialog box for the `dataProvider` parameter of the `MenuBar` component or the `List` component, the label field is prepopulated with "`label0`", "`label1`", "`label2`", etc.

The property `identifier` is optional and if it is not specified then one of the inspectable parameters of the `collectionItem` class will be chosen for you automatically.

Collection Metadata Example

The `MenuBar` example borrows its Collection metadata from `List`, which makes it a bit more complex as an example. The following classes are a very simple example of how Collection metadata works. If you didn't download the entire set of sample files in Part 1 of this article series and you would like to research Collection metadata in more detail, you can download the source files for this example.

Here is the code for `CollectionComponent.as`:

```
package fl.example {  
  
    import flash.display.Sprite;
```



```

    }

    public function clear():void {
        _items = new Array();
    }

    public function getItemAt(index:Number):Object {
        return(_items[index]);
    }

    public function get length():int {
        return _items.length;
    }
}
}

```

And here's the code for ItemExample.as:

```

package fl.example {

    public class ItemExample {

        private var _id:String;
        private var _stringData:String;
        private var _numberData:Number;

        [Inspectable(defaultValue="")]
        public function get id():String {
            return _id;
        }
        public function set id(s:String):void {
            _id = s;
        }

        [Inspectable(defaultValue="string")]
        public function get stringData():String {
            return _stringData;
        }
        public function set stringData(s:String):void {
            _stringData = s;
        }

        [Inspectable(defaultValue="0.0")]
        public function get numberData():Number {
            return _numberData;
        }
        public function set numberData(s:Number):void {
            _numberData = s;
        }

        public function ItemExample() {
            _id = "";
            _stringData = "string";
            _numberData = 0.0;
        }
    }
}

```

```

    }

    public function toString():String {
        return ("[id: \" + _id + "\", stringData: \"\" + _stringData +
\"\", numberData: \" + _numberData + "\"]");
    }
}
}

```

You might notice that in order to make the example very simple, I did not extend `UIComponent`. Since this is not a real component, and it does not really do anything at all, I could get away with this. See the sidebar on Not extending `UIComponent` for more information.

IconFile metadata

`IconFile` metadata is class level metadata. It specifies a PNG file with a custom icon for the Components panel and Library panel to use when displaying the component. `IconFile` metadata only works with SWC-based components. It takes a single nameless value that indicates the path to the PNG file, relative to the FLA file from which the SWC is exported. It looks like this:

```

package {
    ...
    [IconFile("FLVPlayback.png")]
    ...
    public class Foo extends ...
}

```

Since all of the User Interface components and the `MenuBar` component example are FLA-based components, none of them use the `IconFile` metadata. To specify a custom icon for a FLA-based component, use the Component Definition dialog box.

Other metadata

In the User Interface component source code, you will find Style and Event metadata at the class level in many of the ActionScript files. This metadata is not actually used by Flash. It is only used in the automated documentation creation process used by Adobe internally, along with the special comments between `/**` and `*/`. Explaining this metadata, and these special comments, is beyond the scope of this series.

Flash CS3 does not support any other metadata. Any unsupported metadata is ignored without throwing compile errors.

Using the Component Definition dialog box

I had the `Collection` metadata next to my `dataProvider` property definition, but I needed to enter some information in the Component Definition dialog box before the

dataProvider parameter would show up in the Component inspector and Property inspector.

Before opening the Component Definition dialog box, however, I needed to add the ActionScript source files for the User Interface components to my classpath. This is necessary to ensure that the Component Definition dialog box can find the metadata in that source. The path to this source is:

`$(AppConfig)/Component Source/ActionScript 3.0/User Interface`

I modified the classpath in the ActionScript 3.0 Settings dialog box, which is accessed from the Publish Settings dialog box by clicking on the Settings... button next to the ActionScript version checkbox in the Flash tab (see Figure 3).

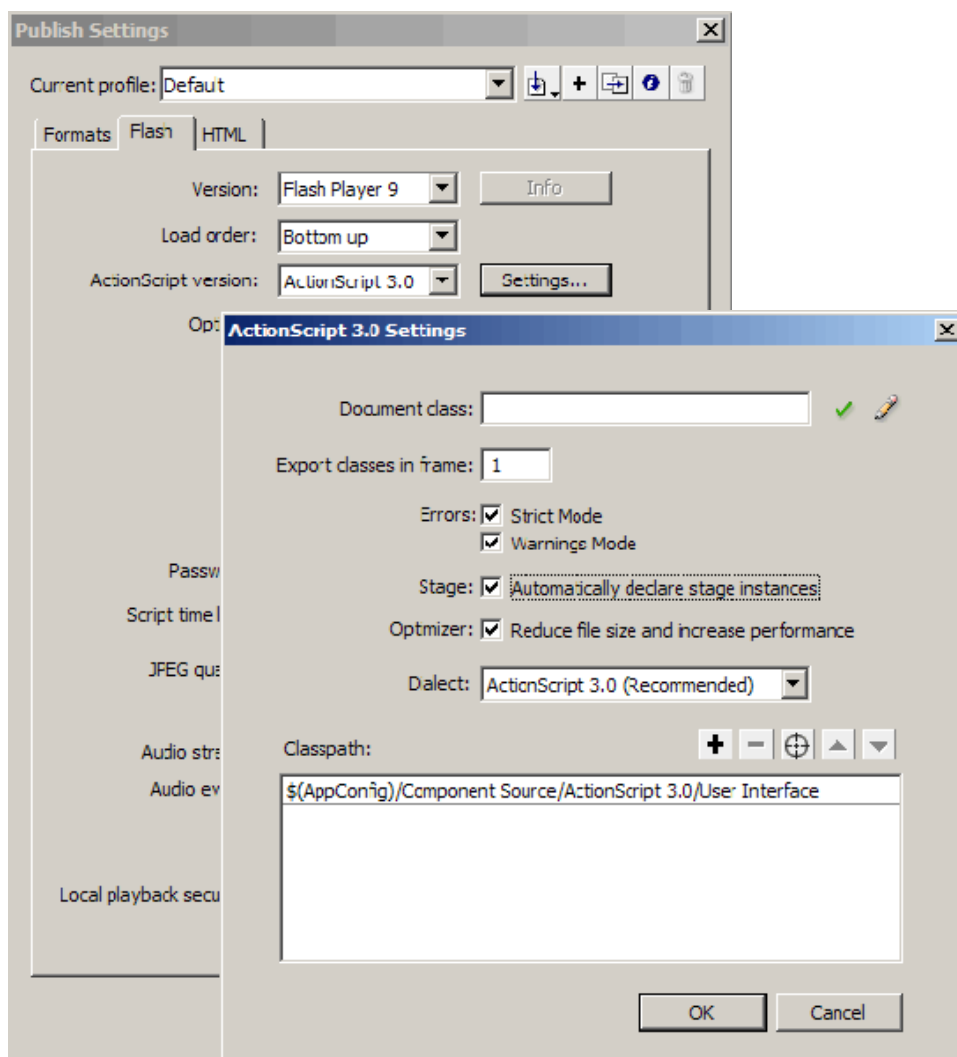


Figure 3. Add the classpath to the User Interface source files in the ActionScript 3.0 Settings dialog box

Now I was ready to set up my component with the Component Definition dialog box. I opened this dialog box by right-clicking on the MenuBar symbol and selecting Component Definition... from the context menu. I customized the following fields in the dialog box:

- I entered fl.example.MenuBar in the Class field.
- I clicked on the icon drop-down menu, under the Description field, and selected the Menu Icon.
- I checked Display in Components panel.
- I checked the Require minimum player version option, which was pre-populated with 9.
- I checked the Require minimum ActionScript version option, which was pre-populated with 3.

Then I clicked OK. When I clicked OK, it took a while for the dialog box to disappear. This is because Flash was processing my source file, fl.example.MenuBar, as well as many of the User Interface Infrastructure source files, looking for metadata. When I opened the dialog box a second time, I saw three parameters listed: dataProvider, enabled and visible (see Figure 4). The dialog box had extracted the information about dataProvider from MenuBar.as and the other properties came from the ActionScript metadata in UIComponent.as. I clicked Cancel this time, so I would not have to wait so long for the dialog box to close, since I did not change any settings.

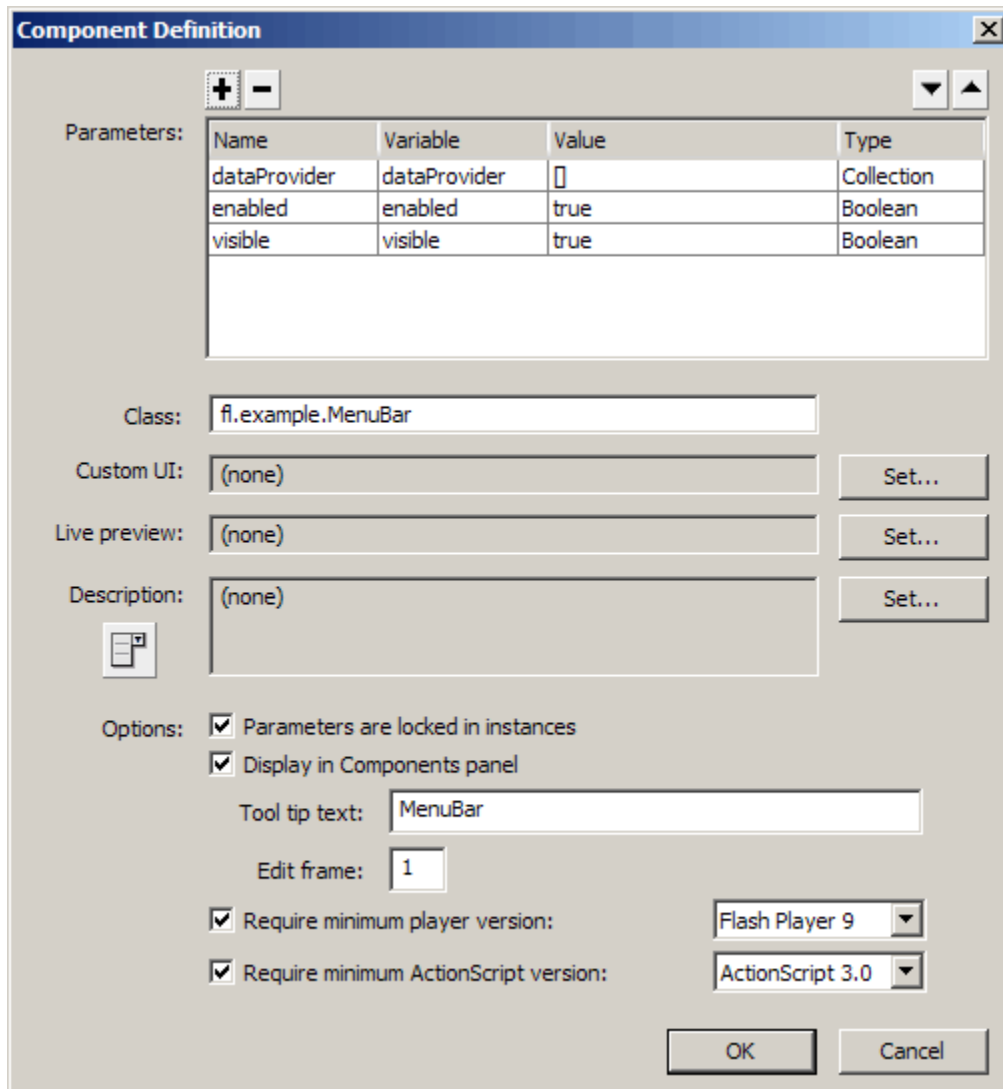


Figure 4. After closing and then reopening the Component Definition dialog box, the three parameters were listed

When I selected the MenuBar instance on the Stage, I could now see my parameters listed in the Components inspector with the default values set (see Figure 5).

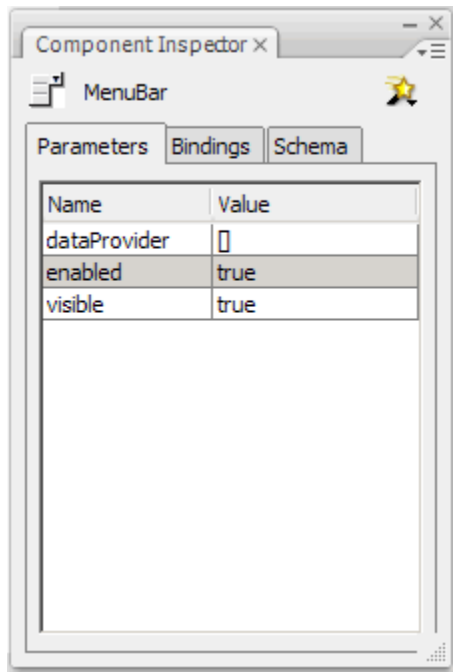


Figure 5. The parameters with their default settings are listed in the Component inspector

Custom icon

For the MenuBar component example I did not create a custom icon, but it is easy to do. Just select Custom... at the bottom of the icon drop-down menu and browse to a PNG file. The PNG file should be small—18x18 pixels is probably the maximum, but you can play around to see how big you can make the icon without it looking horrible. When you select a custom icon, the icon drop-down menu will still show the default component icon, but your custom icon will show up in the Library panel.

My component didn't update!

Sometimes when you make a change in the Component Definition dialog box, whether you are updating the icon, updating the parameters or updating Live Preview, you will not see the changes reflected in the FLA file. Before pulling out your hair, try copying your component into another FLA file and see if this fixes the problem. Alternately, you can save your FLA file, close and reopen it, or better yet do a Save and Compact on your FLA file, to see if that resolves the issue.

Deploying a component to the Components panel

At this point in the project my FLA file could be deployed to the Components panel. Clearly there was a lot more work to do; I did not even have Live Preview yet! But the changes I made in the Component Definition dialog box, specifically checking the

Display in Components panel checkbox option, fulfills the minimum requirements for deployment.

To display MenuBar in the Components panel, I saved MenuBar.fla, copied it into the Components directory, and selected Reload from the Components panel context menu. There are actually two locations for the Components panel: one can be found in the Adobe Flash CS3 application folder and one is located in the user configuration folder. A component installed in the users configuration folder is only available to that user. If you distribute a component as an MXP file that is installed by the Flash Extension Manager, you will need to install the component in the user configuration folder. Assuming a default install path, those folders can be found in the following locations on Windows and Macintosh machines:

Windows	Macintosh
C:\Program Files\Adobe\Adobe Flash CS3\language\Configuration\Components	/Applications/Adobe Flash CS3/Configuration/Components
C:\Documents and Settings\username\Local Settings\Application Data\Adobe\Flash CS3\language\Configuration\Components	/Users/username/Library/Application Support/Adobe/Flash CS3/language/Configuration/Components

With FLA-based components, the name of the FLA file determines the name of the folder in which the component will appear. So I found the MenuBar component in a folder called MenuBar (see Figure 6). I also found the List and TileList components next to it, which makes sense because they were in my Library and of course they also have the option Display in Components panel checked. (You will see how I fixed this later).

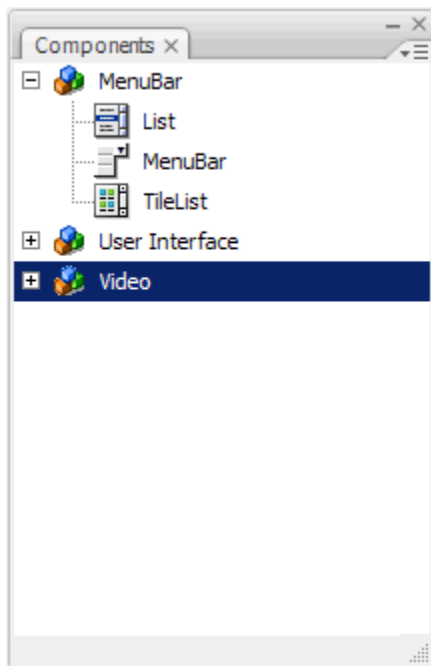


Figure 6. The List, MenuBar and TileList components were listed in the Components panel in the MenuBar folder

When I moused over the MenuBar component, I saw the MenuBar tool tip that I had entered. When I created a new ActionScript 2.0 FLA file, the MenuBar component did not appear at all—since I had entered a required ActionScript version for it.

Please note that you will not be able to overwrite or edit a FLA file in the Components directory once the Components panel has loaded it until you quit Flash, so putting a FLA file into the Components folder is not normally done with work-in-progress components. I did it here just to demonstrate that it would work, but I would not normally bother installing my FLA in the Component panel during development.

Deploying SWC-based components

You would deploy an SWC-based component almost exactly the same way you would deploy a FLA-based component. The SWC-based component is put into one of the two Components folders. If it is placed at the top level of the Components folder, then it will appear in the Components panel in a folder named Standard Components. To customize the folder name, just create a folder with the desired name in the Components folder and put the SWC file in that folder. You cannot nest SWC files more than one folder deep within the Components folder.

SWC-based components can appear in the same folder within the Components panel as FLA-based components by matching the folder name to the FLA-based component's FLA name. For example, in the Video folder there are FLA-based components from Video.fla and there are also SWC-based components that come from the SWCs in the Video folder.

The name of the SWC file is not important. The name of the component will match the name of the movie clip symbol from which the SWC was exported.

You do not need to take any steps to distinguish between ActionScript 2.0 SWC files and ActionScript 3.0 SWC files. Both types can be put side by side in the Components folder and Flash determines the type of SWC and handles each type accordingly.

In contrast to FLA files in the Components directory, a SWC file in the Components directory can be overwritten once it has been loaded in the Components panel. You can repeatedly export a SWC file and select Reload in the Components panel context menu to reload the latest component implementation without quitting Flash and relaunching it.

Implementing Live Preview

The MenuBar component still just looked like a black rectangle on the Stage, so it was time to add Live Preview. With Live Preview, my component would be able to draw itself based on its dimensions and setting information from the Components inspector. A

custom SWF file drives the Live Preview. Luckily, in most cases this custom SWF file does not require any code other than your component code.

To enable Live Preview for MenuBar, I did the following:

- I right-clicked on MenuBar in the Library panel and selected Export SWC File... from the context menu.
- I browsed to the directory that contained my MenuBar.fla and saved MenuBar.swc.
- Outside of Flash, I opened a file browser and browsed to the directory where my FLA and SWC files were located.
- I renamed MenuBar.swc as MenuBar.zip.
- I extracted all the files from MenuBar.zip.
- I moved library.swf, one of the files extracted from MenuBar.zip, into the same directory as MenuBar.fla and MenuBar.zip.
- I renamed library.swf as MenuBarPreview.swf. This is important; do not use the name library.swf when you rename your Live Preview SWF file.
- I went back into Flash, right-clicked on MenuBar in the Library panel and selected Component Definition... from the context menu.
- In the Component Definition dialog box, I clicked on the Set... button next to Live Preview.
- In the Live Preview dialog box, I selected the Live Preview with .swf file embedded in .fla file radio button option.
- I entered MenuBarPreview.swf in the Live Preview .swf file text field.
- I clicked OK in the Live Preview dialog box and in the Component Definition dialog box (see Figure 7).

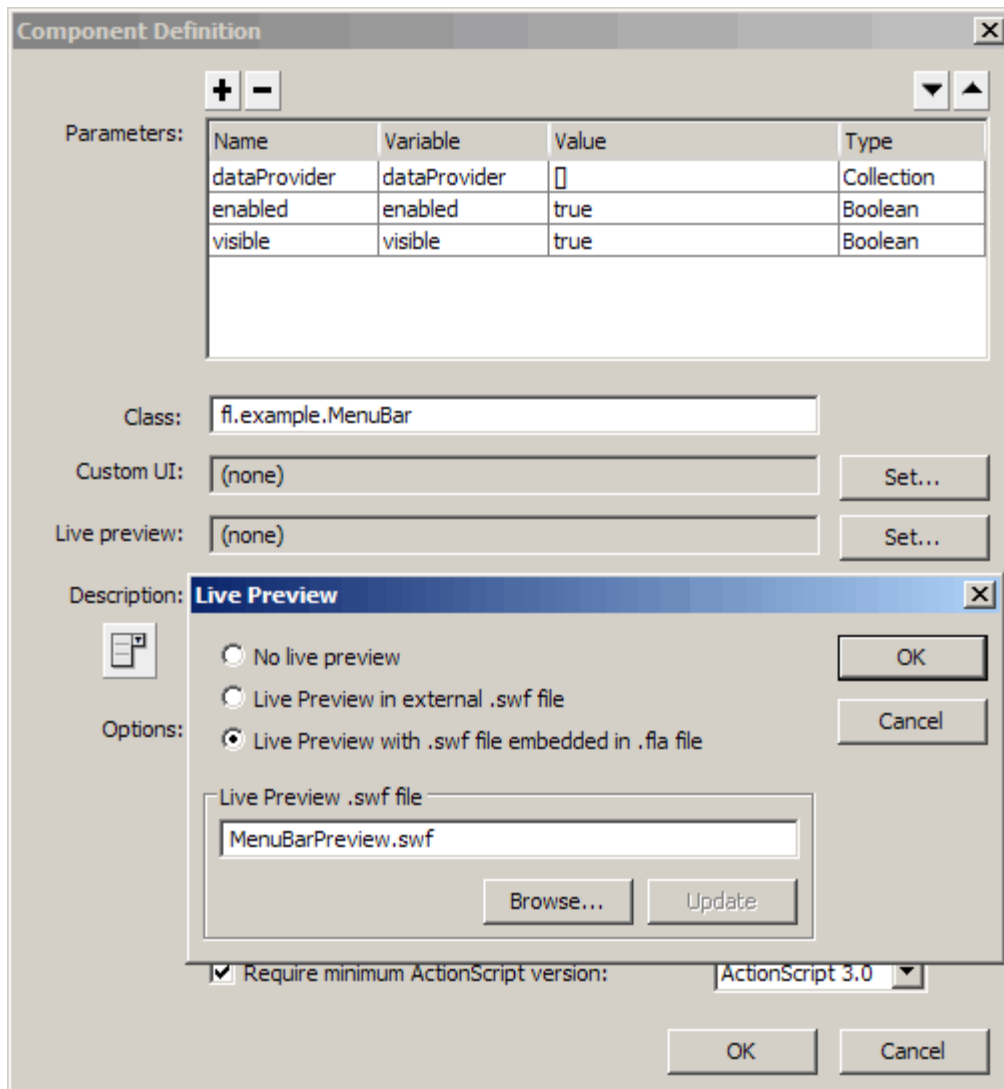


Figure 7. I entered the name MenuBarPreview.swf into the text field in the Live Preview dialog box

After these steps, I had a working Live Preview. It didn't look like much without the dataProvider parameter set—it looked like a gray rectangle instead of a black rectangle. But once I added some menu items in the dataProvider, the Live Preview refreshed right away. I could change dataProvider and MenuBar would update immediately. I could change the dimensions of the component and the menus would resize immediately. And I could test setting the visible property to true and the Live Preview became invisible.

Live Preview for SWC-based components

By default, the Live Preview for an SWC-based component comes from the SWF file that is published by Flash and saved within the SWC. The steps I outlined for creating Live Preview for each User Interface components leverages this fact by generating the

SWC file and extracting the SWF file that would be used for Live Preview if the component were deployed as a SWC-based component.

If you would like to use a custom Live Preview SWF for your SWC-based component, then simply follow the same steps you would follow when creating one for a FLA-based component.

Updating Live Preview

The steps above describe how to set up Live Preview the first time, but after setting it up for the first time you'll need to follow almost identical steps to update the Live Preview whenever you make significant code changes. Follow the same steps, but when you open the Live Preview dialog box from the Component Definition dialog box, the Update button will now be enabled. Click the Update button, and then click OK in the Live Preview dialog box and Component Definition dialog box (see Figure 8).

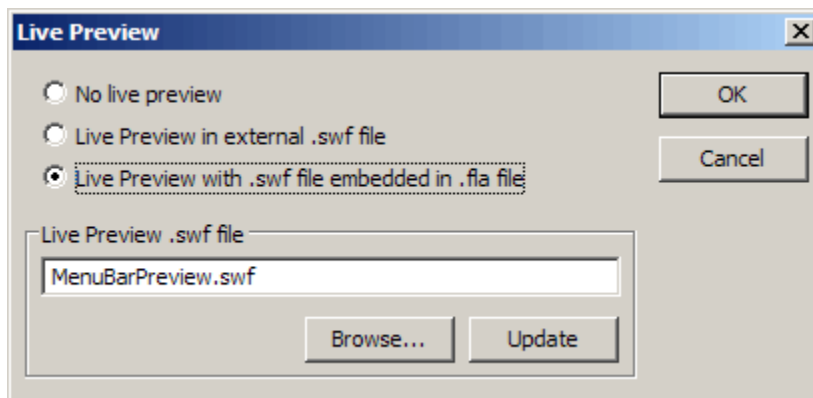


Figure 8. After setting up the Live Preview, subsequent visits to the Live Preview dialog box show the Update button enabled

When following the steps to update Live Preview, always remember that you need to take `library.swf` from the SWC file and rename it. Do **not** use the SWF file from the SWC that already has the correct name, for example `MenuBarPreview.swf`. This file will be the Live Preview SWF that you had previously defined and that you now want to replace. Also remember that you cannot use the name `library.swf` for your Live Preview SWF. Using your component's class name followed by `Preview.swf` is a good naming convention that is followed by all of the User Interface components.

isLivePreview

`UIComponent` declares a protected Boolean variable, `isLivePreview`. In the `UIComponent` implementation of `configUI()`, it initializes the `isLivePreview` to true if the SWF is being used for Live Preview or false if the SWF is not being used. In many cases you will not need this information, and the `MenuBar` component example does not need it, but it can be very useful. For example, the `FLVPlayback` component uses the variable to know whether it should draw the black rectangle with an FLV icon over it or not, and whether

it should use the preview parameter, which only affects Live Preview. In most cases you will not need `isLivePreview` and your component preview should just work.

Creating a custom Live Preview SWF

There are some components that have a Live Preview, which is dramatically different from the runtime functionality. One example of this is the `FLVPlaybackCaptioning` component, which actually does not have any visual representation at runtime but has one for Live Preview so that the user can see it on the Stage. Another example is the `UILoader` component, which at runtime just looks like whatever content it loads, but needs some visual representation on the Stage for positioning purposes. In both of these examples, a separate Live Preview SWF file was used, rather than using `isLivePreview` and putting extra logic into the component source code.

There are a few basic requirements regarding how the FLA file for a custom Live Preview should be created:

- You should set the document class to `fl.livepreview.LivePreviewParent` in the Property inspector.
- You must set the Stage size to match the default size for the component, i.e. the same size as the avatar on Frame 1 of the component movie clip.
- Your FLA file must have only one frame and only one movie clip on the Stage. The movie clip must be placed at (0, 0) and its dimensions should match the stage dimensions.
- You must link the movie clip to a class that implements a `setSize(width:Number, height:Number):void` method that Flash will call when the dimensions of the component change.
- Your linked class should implement setters for any Component inspector parameters that it will use in the preview, but it is not required to implement any of them.

If you create a custom Live Preview SWF file, the steps for connecting it to your component are similar to those for creating a Live Preview SWF file extracted from the SWC file:

- Put your custom Live Preview FLA file in the same directory as your component FLA file.
- Test Movie or Publish the custom Live Preview FLA file.
- In your component FLA file, right-click on the component symbol in the Library panel and select `Component Definition...` from the context menu.
- In the Component Definition dialog box, click the `Set...` button next to Live Preview.
- In the Live Preview dialog box, select the Live Preview with `.swf` file embedded in `.fla` file radio button option.
- Enter the filename of your custom Live Preview SWF file in the Live Preview `.swf` file text field.

- Click OK in the Live Preview dialog box and in the Component Definition dialog box.

When you update your custom Live Preview SWF file, you will need to follow the same steps but will also need to click the Update button in the Live Preview dialog box.

UILoader custom Live Preview SWF

Next, I'll walk you through the creation of the UILoader Live Preview SWF file as an example of how a custom Live Preview SWF is created. If you've downloaded the full set of sample files, you can open the UILoaderLivePreview.zip file – or you can download the source of the UILoader Live Preview SWF to follow along.

The first step was to set the document class to `fl.livepreview.LivePreviewParent`. The classpath did not need to be changed because `LivePreviewParent` is in the default classpath. Next I set the Stage size to 100 pixels x100 pixels to match the UILoader component's default dimensions. At this point, the stage was set up.

Then I created a 100 pixels x 100 pixels black rectangle with a hairline stroke and placed it at the coordinates (0, 0) on the Stage. I selected the rectangle and hit F8 to convert it into a movie clip symbol named Avatar. While the Avatar symbol instance on the Stage was selected, I gave it the instance name `avatar_mc`. I used `avatar_mc` as a placeholder to establish the dimensions of the clip and later I would remove it with ActionScript, just like the component source code does.

Next I clicked F8 again to nest `avatar_mc` inside another movie clip symbol named `UILoaderLivePreview`. I exported this symbol for ActionScript with the same name. `UILoaderLivePreview` will be the class containing the code that drives the Live Preview functionality.

Inside `UILoaderLivePreview`, I created a second layer and created a dynamic text field on the Stage, giving it the instance name `label_txt` and placing it at (0, 0). In the Text Property inspector, I selected the multiline option for the text field, selected the font Arial with a font size of 13 and set the color to dark gray (`#333333`). Then I selected the option to embed all punctuation and basic Latin characters for the font. I will use `label_txt` text field to display the `fl.controls.UILoader` text. I put a dynamic text field on the Stage rather than dynamically creating a `TextField` instance just because it is easier to customize the font, size, color and embed the font through the Text Property inspector than by writing the code.

Notice that unlike a component symbol, I did not use a two-frame structure and I did not put instances other than the avatar on Frame 1. While you could set up a custom Live Preview FLA file like a component, and even have the symbol extend `UIComponent`, usually a custom Live Preview can be pretty simple. Using the Infrastructure in this case might make things unnecessarily complicated.

I was now ready to implement `UILoaderLivePreview`, but first I opened up the ActionScript 3.0 Settings dialog box and unchecked Automatically declare stage instances. This is not required and it is really a matter of style and personal preference. Unlike a component, the custom Live Preview ActionScript code for `UILoader` will only need to work in this FLA file. Note that when you do declare the instances placed on the Stage in your own code, you'll need to declare them as public.

Here is the code for `UILoaderLivePreview`:

```
package {

    import flash.display.*;
    import flash.text.*;

    public dynamic class UILoaderLivePreview extends Sprite {

        private var _width:Number;
        private var _height:Number;

        public var avatar_mc:DisplayObject;
        public var label_txt:TextField;

        public function UILoaderLivePreview() {
            _width = super.width;
            _height = super.height;
            removeChild(avatar_mc);
            draw();
        }

        public function setSize(w:Number, h:Number):void {
            _width = w;
            _height = h;
            draw();
        }

        public override function get width():Number {
            return _width;
        }
        public override function set width(w:Number):void {
            setSize(w, height);
        }

        public override function get height():Number {
            return _height;
        }
        public override function set height(h:Number):void {
            setSize(width, h);
        }

        public function draw():void {
            graphics.clear();
            graphics.beginFill(0xEEEEEE, .8);
            graphics.drawRect(0, 0, _width, _height);
            graphics.endFill();
        }
    }
}
```

```

graphics.lineStyle(1, 0x333333);
graphics.drawRect(0, 0, _width, _height);

// Try full name
label_txt.width = _width - 6;
label_txt.text = "fl.containers.UILoader";
if (label_txt.numLines > 1) {
    label_txt.text = "UILoader";
    if (label_txt.numLines > 1) {
        label_txt.text = "";
    }
}
label_txt.x = Math.max(0, (_width - label_txt.width) >>1);
label_txt.y = Math.max(0, (_height - label_txt.height) >>1);
}
}
}

```

`UILoaderLivePreview` manages its own width and height similarly to the way components do, so it captures the initial width and height values. Then it removes the `avatar_mc` instance after the width and height are known, since the rectangle was only needed as a placeholder to establish the width and height of the movie clip. Finally, the constructor calls the `draw()` method. Since `UILoaderLivePreview` subclasses `Sprite` directly, there is no invalidation model and, although I slightly mimicked it by using a `draw()` method, I needed to call that method directly.

I implemented `setSize()`, which simply changes the width and height and calls `draw()`. I also overrode the `width` and `height` properties, which was not actually necessary but it was trivially simple to do.

The `draw()` method uses `flash.display.Graphics` to draw a rectangle. Then it has some simple logic to help determine whether there is room in `label_txt` for `fl.containers.UILoader`, just `UILoader` or not enough room for any text, and it centers the resulting text in the field.

While I was fine tuning the layout code in `draw()`, I did not publish the SWF file and update `UILoader`'s Live Preview SWF every time I made a change. Instead, I added a second layer to the main Timeline and put code on that layer which called `onResize()` with different widths and heights. Live Preview calls the `onResize()` method when an instance's dimensions change, so directly calling this method emulates what will happen in Live Preview. I had to comment out all of this code before deploying `UILoaderPreview.swf`.

If I had wanted the `UILoader` Live Preview to handle component parameters, I would have simply implemented those parameters. To test the parameters with Test Movie, I would have given an instance name to the `UILoaderLivePreview` instance on the Stage and set properties for it directly using code, similar to the way `onResize()` was called directly.

Using a separate test FLA

For the benefit of the Component Definition dialog box, I added the User Interface ActionScript source to the classpath of MenuBar.fla. An unfortunate side effect of the classpath change is that the Test Movie process for MenuBar.fla is now much slower than it was before! A great solution to this is to start using a separate test FLA file for testing a component as you make code changes to it.

For MenuBar.fla, I created a new FLA file and saved it as test.fla in the same directory as MenuBar.fla. Then I dragged the MenuBar component onto the Stage in test.fla, which brought over all of the dependent symbols, including List, TileList and the entire Component Assets folder into the Library. When I invoked Test Movie from this FLA file, the resulting SWF publishes much faster.

Another advantage of using a separate test FLA file is that it helps avoid spending time on phantom Live Preview or other bugs—which are actually the result of changes to the Component Definition dialog box not being reflected immediately. Trust me, I have spent my share of time doing this before I settled on using the test FLA file workflow.

Whenever I changed any visual assets in Flash, updated the Symbol Properties dialog box, updated the Linkage dialog box or updated the Component Definition dialog box, I always did this in MenuBar.fla. MenuBar.fla is the vehicle for deploying the component; so all changes need to be made there. After making changes, I would drag the MenuBar component from the Library of MenuBar.fla into the Library of test.fla and select Replace existing component in the Resolve Component Conflict dialog box (see Figure 9).



Figure 9. Select the option to Replace existing component in the Resolve Component Conflict dialog box

If you download the component source and examine my test.fla file, you'll see that I like to put some buttons on the Stage and write some simple frame scripts that make simple changes to the component after it has been created. Incorrectly handling updates after a component has been created is a common mistake in writing component code, so performing simple tests like this can catch mistakes early in the development process.

Compiling from source vs. from ComponentShim

The ComponentShim is a compiled clip that holds all of the precompiled byte code for every definition in the User Interface Component Infrastructure. (Well, every class except for the automatically generated classes for skin symbols.) The user does not need the ActionScript files for these classes and other definitions in his classpath because of the ComponentShim.

However, if the user does have the User Interface ActionScript files in his classpath, then the ActionScript 3.0 compiler will prefer the definitions in those files over the precompiled definitions in ComponentShim. A drawback of this is that the compile time is much slower. A benefit is that you can debug the source file and you can make changes to it.

You will see the same result with any compiled clip. For example, if the FLVPlayback component is in your FLA file and you add

```
$(AppConfig)/Component Source/ActionScript 3.0/FLVPlayback
```

to your classpath, you will experience slower compile times. You will also be able to set breakpoints in the FLVPlayback source and debug the code line by line.

Another great workflow when testing small changes you've made to code in a compiled clip is to copy just the ActionScript source files that you are changing into the classpath and change them. That way, you do not have to recompile every file that has not been changed. For example, if you were investigating a workaround for a bug in the DataGrid component, you could copy DataGrid.as next to your FLA file, as always nested within a folder structure that matches the package structure, such as fl/controls. You could then make changes to DataGrid.as and Flash would only compile `DataGrid` from the source code, while all other User Interface Component Infrastructure definitions could come from the ComponentShim. This can save you a lot of time when you need to repeatedly make small changes and Test Movie to review the changes.

Where to go from here

In the next installment of this article series we'll explore adding events to the component, to allow for user interaction. This will allow our MenuBar component to animate at runtime with the collapse and expansion of the drop-down menus.

If you'd like to learn more about working with components, then you may also find these articles helpful:

[Flash and ActionScript components learning guide](#)
[Creating, populating, and resizing a DataGrid](#)
[Displaying images with the TileList component](#)

About the author

Jeff Kameron is a computer scientist at Adobe Systems who has worked on the Flash authoring team since 2002.