

# Creating ActionScript 3.0 components in Flash CS3 Professional – Part 6: The invalidation model

**Jeff Kameron**

Adobe

This is Part 6 of the article series on creating components using ActionScript 3.0. If you skipped the articles leading up to this part, you might want to go back and begin with Part 1 of the series. You'll find a link where you can download the sample files for the entire series in the beginning of Part 1. Or if you would like to just follow along with this section, you can download the sample files for Part 6 below and follow along with this part of the series.

Every component has a `draw()` method, which is responsible for how the component draws itself. Drawing itself may mean actually using the Drawing API in `flash.display.Graphics`, but more often it means dynamically creating instances from the Library, resizing instances, moving instances, making instances visible or invisible and repopulating instances with new data. For example, the Button component's `draw()` method swaps the skin when a user mouses over, away, down and up; it lays out the component when its dimensions change, keeping the label centered; and changes the label when the `label` property is changed.

So what do you think would happen if the following code operated on button instance `foo` in a single frame script?

```
foo.width *= 2;
foo.height *= 2;
foo.label = "New Label";
```

Well, if the `draw()` method was called immediately every time each one of those properties was set, then it would be called three times in a row, instead of just once at the end. This simple example probably would not have a big performance impact, but what if ten properties were set in a frame script? Or, what if 100 rows of data were added to a `DataProvider` in a frame script? You can begin to see how this could get out of hand, and could even cause visible flickering in the display of the component. This is the motivation for working with the invalidation model.

## Using `invalidate()`

Here's how this works. With the invalidation model, instead of calling `draw()` directly when a component's appearance or data changes, the component's code calls the `invalidate()` method. This puts the component instance in an invalid state and schedules a call to `draw()` on the next `enterFrame` event. This way, if three, ten or 100 changes happen all at once, the component will only be drawn once. This is the same `doLater()` approach used in the ActionScript 2.0 components, if you are familiar with

that code. If you are curious about the `doLater()` method you can dig into the component code, but you should never need to use that mechanism yourself; the invalidation model takes care of it for you.

Every time something happens to a component instance that causes its look to change, the instance is invalidated. There are several situations that can cause this: when a component instance receives an event, has a method called or has a property set. Mouse events are probably the most common events that will cause invalidation, but invalidation will also occur if a frame script or other code sets properties that affect dimensions, like `width` or `height`, or changes the functioning of the component, like `dataProvider` or `enabled`.

## Invalidation types and `isInvalid()`

On top of the basic concept of invalidation, there are multiple types of invalidation that are tracked in an instance's invalidation state. The goal is to control the `draw()` method in order to minimize the changes it makes and therefore maximize performance. All the types of invalidation used by the User Interface components are enumerated in `fl.core.InvalidationType`; they are also listed below, along with brief descriptions of the normal causes for each of them:

- `all` - any type of invalidation has occurred
- `size` - dimensions have changed
- `styles` - styles have changed
- `rendererStyles` - renderer styles have changed
- `state` - used when a state has changed. This could be a functional state, such as `enabled`, `selected` or `toggle`, or the mouse state, like `up`, `over` or `down`
- `data` - a new `dataProvider` has been set or the `dataProvider` has been changed
- `scroll` - a user clicked on a scrollbar or the view was scrolled by an API call
- `selected` - the selection in a list-based component has changed

I say the normal causes of the invalidation types because in some cases the types are used in unexpected ways. For example, what sort of invalidation would you expect to use when setting the `label` property on a `Button` instance? I was slightly surprised to see that updating the `label` property adds styles and size to the invalidation state. In addition, your code can use the invalidation states however you want, although you need to keep in mind how the invalidation types will interact with the drawing of your base class. In fact, your component can also define its own types of invalidation outside of this list.

When you call the `invalidate()` method with no arguments, you are implicitly passing the default type, `InvalidationType.ALL`, which covers all types of invalidation, including any that you might define yourself. Your `draw()` uses the method `isInvalid()`, defined by `UIComponent`, to determine whether the current invalidation state includes that type. If the invalidation type `InvalidationType.ALL` is in the state, then `isInvalid()` returns `true` for any type that is passed in. For example, if you called

`invalidate(InvalidationType.ALL)`, any of the following would cases would return a value of `true`:

- `isInvalid(InvalidationType.ALL);`
- `isInvalid(InvalidationType.STYLES);`
- `isInvalid("foo");`

The `isInvalid()` method can be called with one or more types, so for example you could call `isInvalid(InvalidationType.STYLE, InvalidationType.SELECTED)` and it would return `true` if either of those types was in the invalidation state (or if `InvalidationType.ALL` was in the state, of course).

In the `MenuBar` component, I had previously written code to call `invalidate()` when the `dataProvider` is set, when a `dataChange` event is received from the `dataProvider` and when a renderer style is changed, but it was necessary to be more specific about the type of invalidation. The renderer style code, which was largely copied from `SelectableList`, was already specifying an invalidation type with the call `invalidate(InvalidationType.RENDERER_STYLES)`. I updated the `dataProvider` related invalidations to call `invalidate(InvalidationType.DATA)`.

## Using `draw()`

Now it was time to update my `draw` method to limit the updates based on the invalidation type. There are three types of invalidation created by `UIComponent` that every component needs to handle:

- `size` - when the dimensions change
- `state` - when the enabled state changes
- `styles` - when the styles change

In addition, the `MenuBar` component must be updated to handle data and `rendererStyle` invalidation, since it sets those invalidation types on itself. Depending on which class your component extends, you might need to handle other styles as well. The best way to check the types of invalidation that your component's base class uses is to look at the source for its `draw()` method and any methods called by its `draw()` method for calls to `isInvalid()`. Stepping through the component source in the debugger is a great way to find all the `isInvalid()` calls under the `draw()` method.

I put all of the code in my `draw()` method into `if (isInvalid(...))` blocks. In order to do this, I had to rearrange the order of some things.

Check out the code example below:

```
override protected function draw():void {
    // these variables are required for multiple types of
    // invalidation, so I pulled them out of the if blocks.
    // I could have left them declared in the first if block that
```

```

// used them, since AS3 does not do block scoping, but I like
// to write code as though AS3 does block scoping since I find
// it makes the code less confusing, plus many languages do block
// scoping, so coding as though it exists avoids bad habits
var menuBarPadding:Number;
var menuPadding:Number;
var theMenu:List;
var i:int;

// all invalidation blocks set this to true so that drawNow() will
// get called at the end
var needsDrawNow:Boolean = false;

// only destroy and recreate menus if invalidation state has data
// also invalidate all in this case
if (isInvalid(InvalidationType.DATA)) {
    // ensure drawNow() calls happen at the end
    needsDrawNow = true;

    // first clear out everything.
    clearMenus();

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

    // because we destory and recreate the menu Lists, all the styles,
    // renderer styles and layout must be redone, so invalidate ALL.
    // The handling of data change could be made much more efficient
    // by not destroying and recreating all the menus every time,
    // but this example is getting complicated enough as it is.
    invalidate(InvalidationType.ALL, false);
}

// only update styles if invalidation state has styles
// also invalidate size in this case
if (isInvalid(InvalidationType.STYLES)) {
    // ensure drawNow() calls happen at the end
    needsDrawNow = true;

    // set styles
    myMenuBar.setStyle("skin", getStyleValue("menuBarSkin"));
    myMenuBar.setStyle("cellRenderer",
getStyleValue("menuBarCellRenderer"));
    menuBarPadding = getStyleValue("menuBarContentPadding") as Number;
    myMenuBar.setStyle("contentPadding", menuBarPadding);
    myMenuBar.setStyle("disabledAlpha", getStyleValue("disabledAlpha"));

    // set style on each
    var menuSkin:Object = getStyleValue("menuSkin");
    var menuCellRenderer:Object = getStyleValue("menuCellRenderer");
    menuPadding = getStyleValue("menuContentPadding") as Number;
    for (i = 0; i < myMenus.length; i++) {
        theMenu = myMenus[i] as List;
        theMenu.setStyle("skin", menuSkin);
    }
}

```

```

        theMenu.setStyle("cellRenderer", menuCellRenderer);
        theMenu.setStyle("contentPadding", menuPadding);
    }

    // since content padding changes may require layout changes,
invalidate
    // size but do not schedule a draw() on enterFrame, because we will
    // handle it immediately
    invalidate(InvalidationType.SIZE, false);
}

// only update renderer styles if invalidation state has rendererStyles
if (isInvalid(InvalidationType.RENDERER_STYLES)) {
    // ensure drawNow() calls happen at the end
    needsDrawNow = true;

    updateMenuBarRendererStyles();
    updateMenuRendererStyles();
}

if (isInvalid(InvalidationType.SIZE)) {
    // ensure drawNow() calls happen at the end
    needsDrawNow = true;

    // resize the menu bar
    myMenuBar.width = width;
    myMenuBar.height = height;
    // if styles are not invalid this would not have been grabbed above,
    // so grab it now
    if (isNaN(menuBarPadding)) {
        menuBarPadding = getStyleValue("menuBarContentPadding") as
Number;
    }
    myMenuBar.rowHeight = height - (menuBarPadding * 2);

    // 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 - (menuBarPadding * 2)
- 1) / myMenus.length);
        // get menu styles
        if (isNaN(menuPadding)) {
            menuPadding = getStyleValue("menuContentPadding") as Number;
        }
        // make each drop down menu below the corresponding menu bar
cell,

        // make it the same width as the the cell and match its height to
        // its contents
        for (i = 0; i < myMenus.length; i++) {
            // set location and dimensions
            theMenu = myMenus[i] as List;
            theMenu.x = (myMenuBar.columnWidth * i) + menuBarPadding;
            theMenu.y = myMenuBar.height;
            theMenu.width = myMenuBar.columnWidth;
            theMenu.height = (theMenu.dataProvider.length *
theMenu.rowHeight) + (menuPadding * 2);

```

```

    }
}

// add handling for state invalidation to handle enabled/disabled
// toggling,
// which has not worked at all until now
if (isInvalid(InvalidationType.STATE)) {
    // ensure drawNow() calls happen at the end
    needsDrawNow = true;

    myMenuBar.enabled = enabled;
    if (!enabled) {
        closeMenuBar();
    }
}

// drawNow() should be called if any type of invalidation occurred,
// so I had to pull it out of the code path for size invalidation.
// While data and styles invalidation types both force size invalidation,
// the rendererStyles invalidation type does not, but it requires the
// drawNow() call.
if (needsDrawNow) {
    for (i = 0; i < myMenus.length; i++) {
        theMenu = myMenus[i] as List;
        theMenu.drawNow();
    }
    myMenuBar.drawNow();
}

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

```

Notice that the code that destroys and recreates all of the drop-down menu `List` instances is only called when the data type is in the invalidation state. In this case all of the new `List` instances need to be positioned and need styles set on them and everything else, so the code, which handles data invalidation calls, `invalidate(InvalidationType.ALL, false)`. Adding types to the invalidation state in the `draw()` method is a simple technique that can help simplify the logic. Whenever you do this, you should always pass in the second optional parameter as `false` to prevent the scheduling of a `draw()` call on the next `enterFrame` event. The same technique is used in the block handling styling invalidation—which may require layout changes because of the content padding styles—so I added a call to `invalidate(InvalidationType.SIZE, false)`.

Also, notice that I generally tried to avoid making redundant method calls when possible. For example, every `if (isInvalid(...))` block marked a Boolean variable, `needsDrawNow`, to `true`. At the end of the method, I checked this value to see if `drawNow()` needs to be called on the subcomponents. This is a more efficient approach than forcing multiple draws for each subcomponent within the separate `if (isInvalid(...))` blocks. In my reorganization I switched from having a single `for` loop iterate through the

`myMenus` array to having three of these loops. If it was likely that this array would be very long, with hundreds or thousands of entries, this strategy might have caused poor performance. If I anticipated issues like this, I would have updated the logic and made it more complex to avoid potential problems, but since the size of the length of the array in this sample project will never be more than ten, or maybe twenty in any reasonable scenario, writing the code as shown above made the logic easier to follow.

I also added new handling for the `enabled` property, which is triggered by state invalidation. Until now, disabling the `MenuBar` component had not worked properly. Once I updated the code and the `enabled` support was working properly, I generated a new Live Preview and tested that toggling the `enabled` parameter in the Component inspector updated the Live Preview look on the Stage.

## Using component classes without attached Library symbols

When I wrote the code to handle the `enabled` state in the `MenuBar` component, I was just passing through the value of `enabled` to the `MenuBarTileList` instance and letting it handle the rest. Initially the mouse interaction stopped working, but I could not get the `enabled` look to kick in. I added the User Interface ActionScript source files to the `test.fla` classpath and did some debugging. Next, I added some traces, and then I even tried adding a button that called `validateNow()`. Nothing was working, and finally I realized what was happening. It is a problem that you need to be careful to avoid if you work with a class that inherits from `UIComponent` but do not link that class to a movie clip in the Library.

I did not need any of the assets from the `List` component or the `TileList` component, since my `MenuBar` never uses any of their skin symbols, so I removed those symbols from my Library entirely. Then, to avoid using the `ScrollBar` skin assets, I created subclasses of `List` and `TileList`, `MenuList` and `MenuBarTileList`—and these were never linked to Library symbols. Everything seemed to be working fine without the Library symbol connection, since all the assets created dynamically were forced to export for ActionScript by being included in the `MenuBar` movie clip.

However, when I invoked the constructor for `MenuList`, for example, I was essentially creating an empty movie clip. The movie clip didn't have an avatar symbol on the Stage to give it dimensions, so the height and width values were both initially zero. This meant that when the `UIComponent` implementation of `configUI()` grabbed `super.width` and `super.height`, they were both zero. This situation will not necessarily be a problem with all components, but in its `configUI()` method, `BaseScrollPane` creates a shape, which it assigns an alpha determined by the `disabledAlpha` style. It puts the shape above the other display objects to create the disabled look—and if the height and width are both zero, then it fails to draw anything into the shape.

The solution is to create an avatar shape on the Stage before calling `super.configUI()`. I added avatar shapes for both `MenuList` and `MenuBarList`. The updated code for `MenuList` is listed below, with some of the comments removed.

```

package fl.example.menuBarClasses {

    import flash.display.Shape;
    import fl.controls.List;
    import fl.controls.ScrollPolicy;

    public class MenuList extends List {

        override protected function configUI():void {
            // Need to make width and height non-zero or initialization
            // does not work properly. Normal component classes connected
            // to actual symbols in the Library which have the avatar
            // shape or symbol on Frame 1, giving the symbol dimensions
            // before calling super.configUI(), which will grab the height
            // and width and remove the avatar, we check to see if this
            // Sprite has any children, and if it does not then we add
            // a dummy avatar to give it non-zero dimensions
            if (numChildren == 0) {
                var avatar:Shape = new Shape();
                avatar.graphics.lineStyle(0);
                avatar.graphics.drawRect(0, 0, 100, 100);
                addChild(avatar);
            }

            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;
        }
    }
}

```

## Where to go from here

In the next part of this article series we'll cover focus management and how to control which part of the component is active when mouse or key press events are encountered. I'll discuss working with FocusManager and how it interacts with display objects.

As I've mentioned many times in this article series, the ActionScript 3.0 debugger is an essential tool to help you troubleshoot issues with your projects. If you are familiar with the previous version of the debugger for ActionScript 2.0, you'll be excited to see that

there are many new improvements to the debugger for ActionScript 3.0. To learn more about the debugger included with Flash CS3, and to ensure that you are using this resource to its full potential, be sure to check out this helpful Developer Center article:

[Introducing the ActionScript 3.0 debugger](#)

**About the author**

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