FlexJS Component Source Code Patterns

This page is not about "coding conventions" like where to put your curly braces.

This page is about how a single .as file produces a component that runs in a SWF as well as JS in the browser (or other JS runtime).

Introduction

Here are some things to keep in mind when writing components for FlexJS. Note that these considerations should not be needed for developing applications for FlexJS as the components should abstract away these issues so application developers don't have to know about them.
Component development should have some similarities to developing components for the Flex SDK. You create a Library Project and/or use compc to compile it into a SWC. You'll probably extend some base class. You still need Event metadata if you are dispatching events.
The main difference is that there are different base classes and events, but also: you may need to write code that only works in the SWF or in JavaScript in the browser. If you are lucky, your component will not need platform-specific code. In fact, many components like DataGrid and Charts have no platform-specific code. The platform-specific stuff tends to be in the low-level components like Button. In more detail:

Conditional Compilation

We use conditional compilation to specify platform-specific code so that both the SWF and JS versions can be compiled from the same .as file. Thus there is only one org.apache.flex.html.Button.as that gets compiled into a SWF/SWC and cross-compiled into a .JS (and included in the same SWC for use by FalconJX).
We have two conditional compilation flags: COMPILE::SWF and COMPILE::JS

Base Classes

SWF UI classes do not have one common UI base class. That's because Flash has flash.display.Sprite, flash.text.TextField, and flash.display.SimpleButton all of which have different API surfaces. Yes, they all extend flash.display.DisplayObject, but org.apache.flex.html.Button does not extend org.apache.flex.html.UIBase like org.apache.flex.html.List does. You should not use DisplayObject in code you want to cross-compile, so the only common type to reference "any" FlexJS component is the interface org.apache.flex.core.IUIBase.
The JS version of UI classes almost all extend org.apache.flex.core.UIBase.

Different inheritance

The section on Base Classes, plus the desire to abstract away platform differences, means that the JS version of a class may extend a different base class than the SWF version.
For example, for SWF, org.apache.flex.html.Button extends org.apache.flex.core.UIButtonBase, but in JS, it extends org.apache.flex.core.UIBase.
The source code for Button and other components with platform-dependent base classes is to simply use conditional compilation to put two class definitions in the same .as file. So, for Button.as, it will start with something like:

```
/**
 * class documentation
 */
COMPILE::SWF
public class Button extends UIButtonBase
{
}
…
COMPILE::JS
Public class Button extends UIBase
{
}
```

When examining a source file, if you see a COMPILE::SWF flag on the line before "public class", you are pretty much guaranteed to find a COMPILE::JS flag later in the file.

Same Inheritance

The majority of UI components extend UIBase both for SWF and JS versions. In those cases, there is no COMPILE:SWF flag on the line before "public class". Instead COMPILE::SWF and COMPILE::JS flags are scattered throughout the source file. There are two patterns of usage:
1: Per-definition. Entire functions and vars (but not import statements) can be controlled by a conditional compilation flag of the form:

```
COMPILE::SWF
public var someVar:Object;


COMPILE::SWF
public function someFunction():void
{
}
```

2: Per-block. Individual or groups of import statements and chunks of code in method bodies can be controlled by a conditional compilation flag of the form:

```
COMPILE::SWF
{
  import somePackage.someClass;
}


public function foo():String
{
    COMPILE::SWF
    {
       return “hi”;
    }
    COMPILE::JS
    {
        return “bye”;
    }
}
```

Note that you will not get an error that a method like this doesn’t return a value because the conditional compilation essentially happens during the parsing phase so the compiler thinks the method looks like:

```
public function foo():String
{
    return “hi”;
}
```

Or

```
public function foo():String
{

    return “bye”;
}
```

JS Components Wrap HTMLElement, are DisplayObjects.

It has rarely been an issue, but keep in mind that in the SWF, the component IS the DOM object, but in JS, the component “wraps” or references the DOM object (an HTMLElement). In theory, the APIs in the base classes for addEventListener, dispatchEvent and setting x,y, width, height abstract these differences away.

Wrap platform code if you can

When looking at all of the places where you have to inject conditional compilation, consider whether you can introduce a new helper class that wraps just enough platform-specific code that the main component will have no or much less conditional compilation that it would otherwise. Examples are org.apache.flex.core.ApplicationBase, which greatly simplified org.apache.flex.core.Application, and org.apache.flex.effects.PlatformWiper which abstracts how to occlude a display object so the Wipe effects have no conditional compilation code in them.

Casting

You may find it annoying at times to have to add types to vars in JavaScript being ported back to ActionScript. And once you do, you will start to get errors about missing properties. Well, that’s the trade-off of type-checking. Some time later you realize it will save you from leaving some other bug in the code. In COMPILE::JS blocks you will find, for example, code that could look like:

```
element = document.createElement(“div”);
element.flexjs_wrapper = this;
```

But element is of type WrappedHTMLElement (so you can set the flexjs_wrapper property without an error) and document.createElement returns HTMLElement so you get an error and have to cast. Try to use “as” casting as in:

```
element = document.createElement(“div”) as WrappedHTMLElement;
```

Now if you think about it, and know how “as” works in JS, you’ll realize that the “as” should fail in JS. That’s because “as” works by examining data structures attached to the FlexJS classes that the actual HTMLElement won’t have. So you have to do one more thing, which is tell the compiler to not generate the “as” code by adding in the asdoc for the method:

```
@flexjsignorecoercion org.apache.flex.core.WrappedHTMLElement
```

You’ll see in the code @flexjsignorecoercion for native HTMLElements as well. There might be a better way to do this, but that’s what we’re using for now.

JS-only or SWF-only Components

Sometimes a component only needs to exist for the SWF version and the class dependencies don't require that a class is defined for the JS version (and vice-versa). An example is the ScrollBar-related code. The SWF versions of Container and List are capable of using ScrollBar-capable views but the JS version of Container and List and their views don't reference Scrollbars because they are built into the browser's DIV (the JS views simply toggle ScrollBar-related styles on the DIV). Same for Borders and Backgrounds. For cases like these, the components have so far never been in the MXML manifest and are only listed in the XXXClasses file (for Core.swc, CoreClasses.as). The pattern to use is to use COMPILE::SWF or COMPILE::JS around the line for that component in the XXXClasses file. If you use the "Different Inheritance" pattern above, it can cause errors as then the source file actually looks empty to the compiler. So the best option is not to compile it at all. Below is an excerpt from CoreClasses.as.

```
    import org.apache.flex.utils.EffectTimer; EffectTimer;
    import org.apache.flex.utils.MixinManager; MixinManager;
        COMPILE::SWF
        {
            import org.apache.flex.utils.PNGEncoder; PNGEncoder;
            import org.apache.flex.utils.SolidBorderUtil; SolidBorderUtil;
            import org.apache.flex.utils.StringTrimmer; StringTrimmer;
    }
        import org.apache.flex.utils.Timer; Timer;
    import org.apache.flex.utils.CSSUtils; CSSUtils;
        COMPILE::JS
        {
            import org.apache.flex.utils.Language; Language;
        }
```

In this snippet you can see that PNGEncoder, SolidBorderUtil and StringTrimmer are only needed for SWF versions, and org.apache.flex.utils.Language is only used in JS versions.

SWF- or JS-specific APIs

Each SWC project is allowed to have public APIs that are used for communicating with other SWCs and not for use by the application developer. For example, the JS implementations may want to address the Window object directly. While it is possible to wrap-up and abstract away these differences between the SWF and JS runtimes, those abstractions can have performance issues, so instead we are currently opting to allow differences in public APIs based on the runtime.

So, each SWC project actually produces two SWCs. One we call the "JS" SWC that may have JS-specific APIs, and the main SWC that has the byte code for the SWF and the cross-compiled JS files and can have SWF-specific APIs. A future task is to upgrade the documentation system to mark certain APIs as runtime specific so the application developer knows which APIs will exist on both runtimes.

The "JS" SWC is identified by having "JS" appended to the SWC name, so there is a Core.swc that contains SWF byte code and cross-compiled JS files, and a CoreJS.swc that contains the JS-specific APIs. Downstream SWC projects then use Core.swc for building the main SWC and CoreJS.swc for cross-compiling and building its own "JS" sec for its downstream customers.

Build-related Files

There are several files involved in building the main SWC. The goal is to cross-compile the JS version, then compile the SWF version and create a SWC that includes the .JS files from the cross-compile.

For each SWC project, there will be a frameworks/projects folder (i.e. frameworks/projects/HTML) and a frameworks/js/FlexJS/projects/ folder (i.e. frameworks/js/FlexJS/projects/HTML). In addition, there are two output folders of SWCs. The main SWCs go in frameworks/libs, the "JS" SWCs go in frameworks/js/libs.

The process for building all of the SWCs is to first build the "JS" SWCs and cross-compile the AS to JS with COMPILE::SWF=false and COMPILE::JS=true. This generates the SWCs that go in frameworks/js/libs, and target/generated-sources folders in each of the projects in frameworks/js/FlexJS/projects.

The next step is to build the main SWFs. The builds compile the AS with COMPILE::SWF=true and COMPILE::JS=false. They package up the JS files from target/generated-sources in the project's frameworks/js/FlexJS/projects folder.

The build.xml file for a "JS" SWC should have two main targets:

1. compile-asjs: This cross-compiles the JS version using the js.swc from Falcon (as opposed to playerglobal.swc or airglobal.swc) for the "built-in" classes (Object, Array, String, Number, etc). The COMPILE::SWF flag should be false and the COMPILE::JS flag should be true.
2. compile-js-swc: This compiles a SWF for a SWC but still using the js.swc from Falcon. This SWC is placed in an js/libs folder for use as a dependent SWC by downstream SWCs. This SWC is needed so that only the API surfaces exposed by COMPILE::JS are available to consumers of this SWC. Application developers should never need to access these SWCs, only downstream SWC developers.

The build.xml file for a SWC should have 3 main steps:

1. compile: This is the final phase that compiles a SWF for a SWC with COMPILE::SWF true and COMPILE::JS false and using playerglobal.swc or airglobal.swc for the built-in classes.

The list of files that are compiled by each phase is controlled by a compile-js-config.xml file for steps 1 and 2, and compile-as-config.xml for step 3.

The compile-js-config.xml file

- Should have an empty external-library-path entry. The external-library-path is specified in build.xml so we can use Ant to resolve the reference to js.swc in the Falcon repo
- Should use upstream SWCs from the js/libs folder, not the frameworks/libs folder (so that the API surfaces are the ones exposed by COMPILE::JS

The compile-as-config.xml file

- Should have playerglobal.swc or airglobal.swc as the external-library-path entry
- Should use upstream SWCs from the frameworks/libs folder and not the js/libs folder
- Should have "include-file" entries for the js/FlexJS/projects/<projectname>/target/generated-sources folder

```
<include-file>
    <name>js/out/*</name>
    <path>../../../../../js/FlexJS/projects/<projectname>JS/target/generated-sources/flexjs/*<</path>
</include-file>
```

The build.xml not only includes the JS files in the JS SWC folder, but also copies them to frameworks/js/FlexJS/generated-sources. The FalconJX compiler is told to look at frameworks/js/FlexJS/generated-sources first, then look in the SWCs. This is done to enable folks to monkey-patch JS files in frameworks/js/FlexJS/generated-sources and not have to rebuild a SWC. The danger is that a stale file in frameworks/js/FlexJS/generated-sources can override your changes that went into the SWC.