Lassie Shepherd Library Tutorial

Introduction

What is a Lassie Library?

A library is a compiled SWF movie with all of its embedded graphics, animations, and sounds configured in a way that is accessible by the Lassie engine. So, think of a library SWF as just a box of assets (images, sounds, etc). Lassie will never display the box, it will just extract its contents and use them to render a game. Since an asset library SWF is never directly displayed, it needs nothing on its main timeline except for a list of what library symbols are available within it.

The Advantage of Runtime Library Sharing

Flash (AS2) previously restricted access to SWF libraries, making true dynamic library sharing impossible. Because of this, previous Lassie builds forced every media element to load in as an individual SWF movie. Even when multiple objects in a room used the same SWF file, each object would have to load the SWF separately. This was especially burdensome for large SWF assets like a game's character movie, which would have to reload each time a new room was viewed, rather than retaining and reusing the single loaded character asset.

AS3 has thankfully made library sharing possible. Now graphics, animation and sound can be compiled into a single SWF movie, loaded once, then all those assets can be extracted and used by Lassie as if they were part of Lassie's own library. So, this allows smarter asset management and easier loading. Where a Lassie room in a previous build of the engine might have needed to make 50 separate file requests to pull all the assets required for a single room, Lassie can now load a single library SWF for each room and extract all the room's resources from it. Libraries can also be loaded by the game as a whole, making their resources available to all rooms in the game without any redundant loading.

Required Skill Level

Do you know how to use/manage MovieClips within your Flash library? If the answer is "no", then you've got some very basic Flash skills to learn before getting into Lassie. If the answer is "yes", then fear not! You'll be creating, editing, and managing MovieClips just the same as you always do within Flash; the only difference is that you'll need to tag each clip with a unique identifier, called a "class", then list those identifiers on the first frame of your movie where Lassie can read them. That's it.

Tutorial Vocabulary

There will be several terms used commonly in this tutorial that you should be familiar with before proceeding. Most are pretty basic Flash terms and concepts.

- **Document (or "FLA")**
  This refers to a Flash-native project file with the file extension ".fla".

- **Document Library**
  Basically, think of this as the Flash Library Panel.

- **Library SWF**
  This is what we're trying to create. This is a compiled SWF movie that Lassie will load and pull media from. The point of this tutorial is to explain the process involved in configuring a library SWF.

- **Symbol**
  A "symbol" is anything contained within the document library. The Flash Library Panel shows a list of symbols available for use within the document. Common symbol types are MovieClips, Graphics, and SimpleButtons. Lassie cannot access library symbols until they have been "classified" (keep reading for more details).

- **Symbol Name**
  This is a symbol's name within the document library. It is used exclusively for human-legibility while browsing the Flash Library Panel. It has no technical use or value, so it is important not to confuse a symbol name with a "class name" (keep reading for details).

- **Class**
  Think of this as an "elevated state" of a library symbol. Once a symbol is given a class, it becomes accessible to Lassie.

- **Class Name**
  This is the name given to a class. However, unlike a symbol name, the class name is of great technical importance. The class name is the identifier that Lassie will use to reference and extract an asset from the library SWF.

For Advanced Flash Users

If you have some advanced experience with Flash, you'll probably notice that some of the steps in this tutorial explain the simplest solution, not the most organized, elegant, or best technique. As such, please feel free to build upon this tutorial's basic concepts if you have the background to do so. To help advanced users, there are some notes included in this tutorial that specifically target developers with a broader knowledge of Flash and ActionScript 3. If an advanced user note does not make sense to you, don't worry. Move on and don't trouble yourself with the technical details.

Library Types and Uses

As previously indicated, Lassie uses libraries in several manners. Uses include:

Game Libraries

Required, minimum of one. This library (or set of libraries) is loaded once at the beginning of the game. Game libraries must include all avatar character artwork, inventory items, interface elements, or any other resource that is used globally throughout the entire game. This library (or libraries) may contain common assets that are used by various rooms within the game as well; however, consider the "common libraries" tactic (outlined below) before overloading you main game library with room-specific artwork.

Room Libraries

Suggested, minimum of one per room. While room assets could technically be accessed from game libraries, that would place massive overhead in the initial game load. Instead, it's suggested that you create an individual library for each room within your game that will load once before the room's first viewing. Loaded libraries will remain in memory until specifically told to unload, so a room will not be required to reload again.

Voice Libraries

Suggested. Voice libraries contain sound media which is used for "talkie" game dialogue. While voice sounds can technically be included in the primary libraries used to construct a game, they weigh down the game with excessive file weight that is wasted if in-game voices are disabled. So, voice libraries allow you to store all voice media in separate, designated libraries which are only loaded into a game when voices are enabled. Voice libraries can be defined globally or for individual rooms within the Shepherd editor.

Common Libraries

Suggested general practice. This is not so much a specific library use as it is just a practical concept. When you have common elements that are used in multiple rooms (though not used frequently enough to merit be loaded as part of the game as a whole), just place the elements into a single library that is loaded into multiple rooms. The assets will be loaded the first time they are needed, then will remain in memory for all other instances that consumes them. That's more mileage with less loading.

Creation Process

Step 1: Create a New AS3 Flash Document

Open Flash CS3 (or later) and select “File > New...” from the application menu. The resulting window will allow you to select a type of document to create. Select "Flash File (ActionScript 3.0)", and click OK. The ActionScript 3 setting is mandatory for the proceeding steps to work.

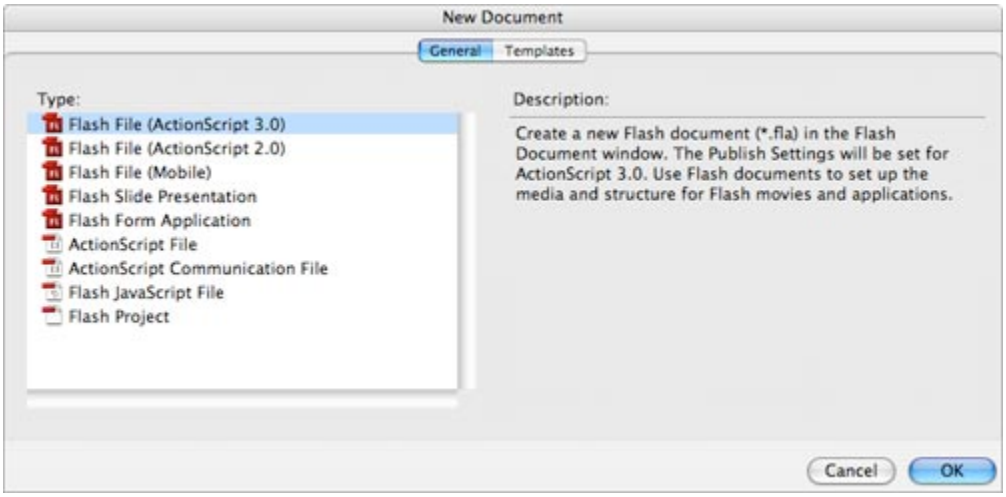

fig 1.1: Creating a new ActionScript 3.0 document

Save this new Flash document into the provided Lassie resources folder containing the Lassie "com" directory. For basic library configuration, the FLA document must be saved into the same folder as the Lassie "com" directory, otherwise resource links will not work. Be sure to save the new FLA document before proceeding.

Note to advanced Flash users: If you'd like to organize your library FLA's into a neat directory structure, specify a class path for the document that points to Lassie's "com" folder then place the document into any directory structure that you'd like.

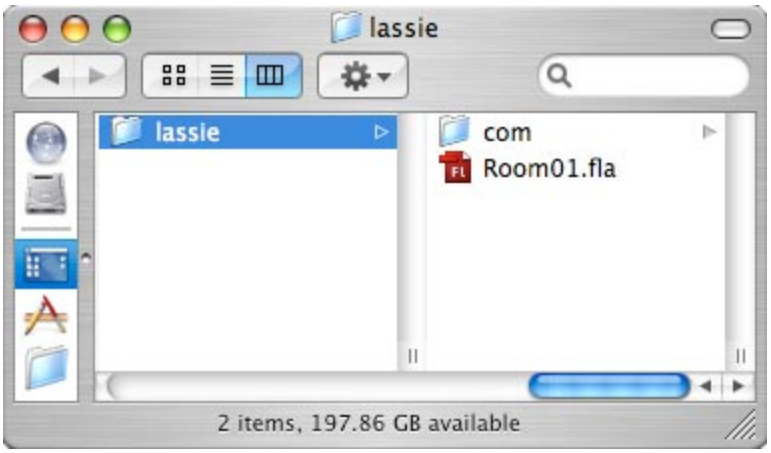

fig 1.2: The new Flash document should be saved in the same directory as the provided Lassie "com" directory.

Step 2: Set the Document Class

Click anywhere on the blank document's stage to bring up the document's properties in the Flash Properties Panel (Window > Properties > Properties). Assuming that the document was created as a ActionScript 3 file, there will be property field called "Document class". Enter the following path into the "Document class" field:

```
com.lassie.lib.Library
```

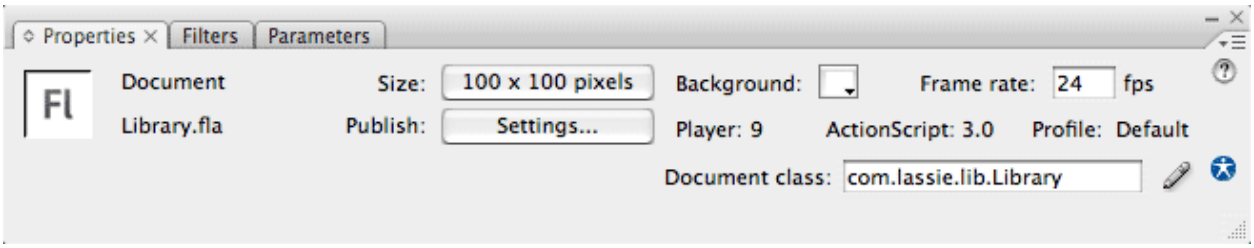

fig 2-1: Set the document class within the Flash Properties panel.

That path points to a provided script file that will configure the document for use within Lassie. You do not need to alter that script file in any way. Just make sure you reference it properly. The above link will work assuming that your Flash document is saved (emphasis: SAVED) into the Lassie directory containing the "com" folder.

Note to advanced Flash users: it is advised that you do not subclass the document.

Step 3: Go About Your Business

Here's where you go about your Flash work as normal. Create library symbols (ie: MovieClips and Sounds), import artwork, design layouts, create animations, etc, just like you always do within Flash. The only trick here is to not place anything on the main (root) timeline of the document. For any and all timeline work, just create a new MovieClip symbol within the library and work on that clip's timeline.

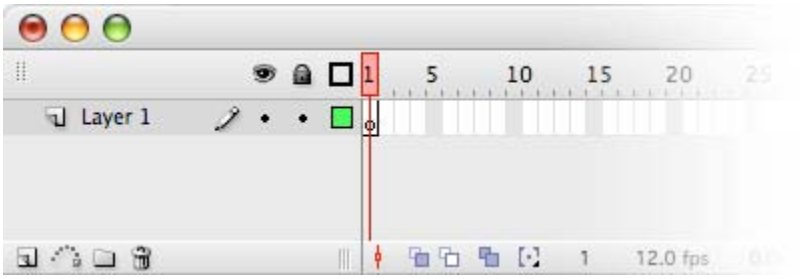

fig 3-1: Document library and document's main (root) timeline.

As your game media develops, it should all exist as just library symbols. If you want to place media onto the document stage while you edit it, that's fine. Just remove it again from the stage when you're finished editing. Ultimately, the document should remain with nothing but a single blank keyframe on its main timeline, just like it was when the document was created. All document assets just exist as symbols within the library.

Step 4: Set Symbol Classes

In order for library symbols to be extracted and used by Lassie, the symbols must be tagged with a custom identifier defining their "class". You don't need to worry about the technical details of what classes are or how they work, you just need to know the rules and steps of defining them. Once you have given a library symbol a class name (and completed steps 5 and 6), that symbol class will become available within the Lassie engine for use within your game. Non-classified symbols will not appear as options for use within the Shepherd editor.

Symbol Names Versus Class Names

Go into the Flash Library Panel and you'll see a list of all library symbols. The name displayed for each item is its "symbol name", which unfortunately (for the sake of a simple explanation) is meaningless; what we need to define are "class names". This is a slightly confusing aspect of Flash symbols: what a symbol is called and what ActionScript references it as are two different things. Here's the breakdown:

Symbol name = Flash library panel title = **meaningless.**
Class name = ActionScript identifier = **IMPORTANT.**

Thankfully, there is no harm in just naming a symbol and its class the same thing, so for the sake of ease and clarity... DO THAT. It makes life a lot easier when you can think of a symbol and its class by the same name, rather than keeping track of what symbol name goes with what class name.

Naming Classes

As we proceed with creating classes, it's important to be aware that we are (in essence) weaving new elements into the fabric of the Flash document structure. As a result, we need to step lightly and make sure we don't accidentally use Flash-specific names that will conflict with the programming of Flash its self (or Lassie for that matter). We also need to make sure that each class name we specify is valid. Don't worry, this sounds a lot scarier than it really is. Follow some simple guidelines and you shouldn't have any problems. Here are some rules to follow:

- Class names are all one word and limited to characters a-z, A-Z, 0-9, and underscore. Also, a class name cannot start with a number as the first character.
- Capitalize class names. Capitalizing the first letter of a class name will automatically isolate it from program properties (which start with lowercase letters). Example: "GoodClassName" verses "badclassname".
- Use mixed case (also known as "camel-case") to separate words. Example: "GoodClassName" versus "Badclassname".
- Be as specific as possible. To use a broad name like "Mouse" risks matching a Flash-native class name. So, personalize your names with information specific to your project, like "SkippyTheMouse". A good practice would be to start all room-specific objects with the room's name, such as "Room01Bg", and "Room01Table". The more detailed you make the class name, the less likely you will be to encounter a name conflict.
- Class names must be unique. Just like avoiding naming conflicts with Flash, we also need to avoid conflicts between our own class definitions. This is really easy to avoid because Flash will pop an alert message if you try to give two classes the same name.

What to Classify

By giving a library symbol a class, we are enabling ActionScript to extract it from the library and use it within the Lassie engine. However, that does not mean that every single media element in your library needs to be classified for ActionScript. You could drive yourself crazy by specifying that many class definitions! All you need to classify are the primary library symbols that will be used by your game; you DO NOT need to classify all the individual parts and pieces that compose them. Let's look at an example...

The Lassie character sprite is a MovieClip with a frame for each walk cycle, talk animation, rest state, etc. On each one of those frames is a nested MovieClip depicting the specific character animation. So, we'll think of the main character MovieClip (with all the animated states) as the "parent" movie, and each individual character animation placed inside it as "child" MovieClips. In short: the "child" MovieClips are nested inside of the "parent" MovieClip.

When exporting the character MovieClip, the only symbol that would need a class name would be the "parent" clip (or main character movie). All of its child animations will automatically be included with it. The only reason that you would need to classify one of the child movies would be if that individual animation is ever needed by its self (out of context of the parent).

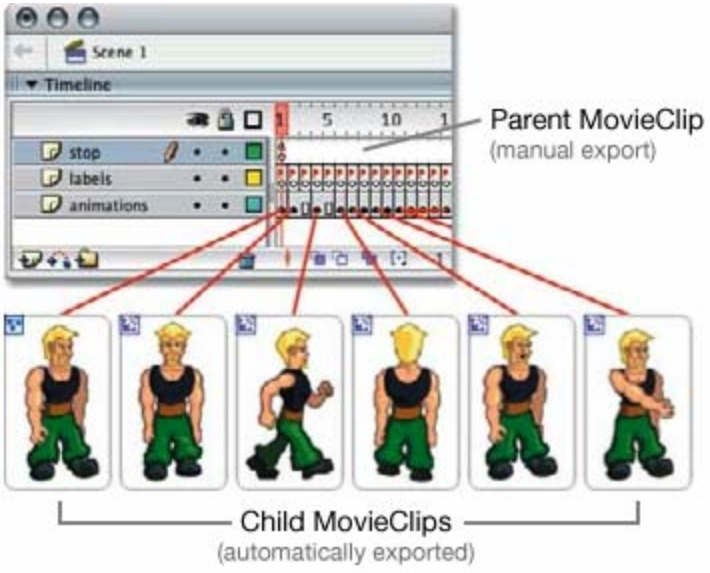

fig 4-1: A "parent" MovieClip timeline with "child" movies placed on it. Exporting the parent will automatically include the children.

Here are some simple guidelines for choosing media to export:

- **Art:** Only classify MovieClips. DO NOT classify bitmaps, graphic symbols, or simple buttons by themselves. If you ever need a non-MovieClip media type placed into your game, just nest it in a new MovieClip and classify that.

- **Art:** Only classify MovieClips that are COMPLETE compositions. Do not classify the individual parts and pieces that make up the MovieClip unless the child element is a complete composition that will be used elsewhere by its self.

- **Sound:** Only classify Sound media.

Setting a Library Symbol's Class Definition

Now that we're up to speed on the basics of how to use classes, you'll be happy to know that the actual process of defining a class is pretty simple. Start by going into the Flash library and right-click on a MovieClip or Sound symbol to classify. From the context menu that pops up, select "Linkage".

Update for Flash CS4+ users: The "Linkage" option has been removed from the library options menu. Instead, select the "Properties..." menu option, and toggle open "Advanced Settings" within the properties window that pops up.

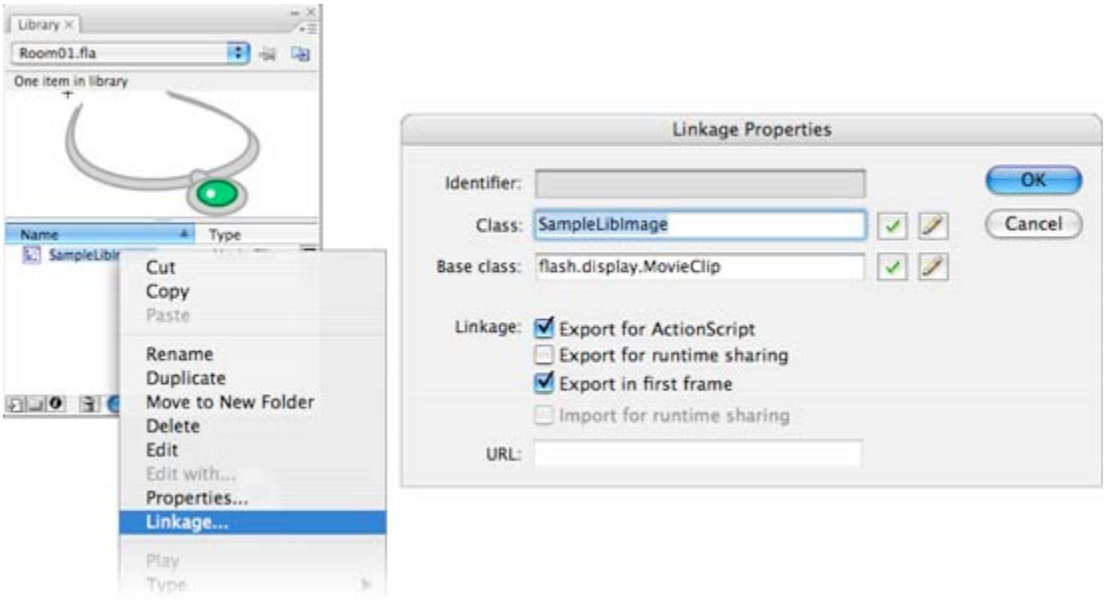

fig 4-2: Opening and configuring the linkage properties window.

A windoid will open with some options. First, check the "Export for ActionScript" option, then make sure the "Export in first frame" option is checked (it should be checked by default). Do NOT check "Enable for runtime sharing". Finally, enter a unique class name in the "Class" field and click "OK". Again, your job will be much easier if you make the class name match the symbol name. You'll also notice a field called "Base class". The symbol's base class should already be configured correctly by Flash as follows:

When classifying MovieClips, "Base class" should be set to:

```
flash.display.MovieClip
```

When classifying Sounds, "Base class" should be set to:

```
flash.media.Sound
```

If the symbol's base class is NOT automatically set to one of those two values, then chances are that you're trying to classify something that you shouldn't. Keep in mind that only MovieClips should be classified as art assets. Other Flash graphics like Bitmaps or SimpleButtons should be nested into a MovieClip symbol for use within Lassie.

Step 5: Log Your Classes

Home stretch! You've now created media classes that can be used within the Lassie engine; however, Lassie does not know that a class is available within the library without having the class logged into the library registry (ie: a record of all classes that exists within the SWF library). To log your library's classes, go to the main (root) document timeline where you should (conveniently) find a single blank key frame waiting. If there's more on the main timeline than a single blank keyframe, then go back and review step-3.

Select that single blank keyframe on the main timeline and open the Flash Actions panel (Window > Actions). Within that frame's Actions panel you'll need to log all classes that you've created in your library using these two script commands:

```
// Log a MovieClip class
addMovieClip( MyMovieClip );

// Log a Sound class
addSound( MySound );
```

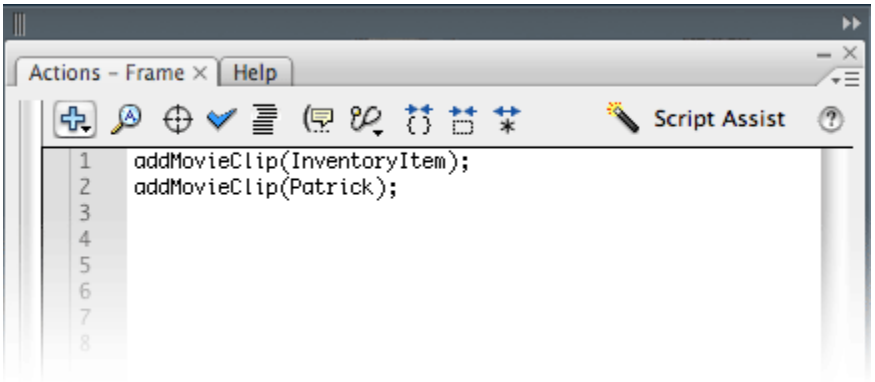

fig 5-1: Logging class identifiers within the Frame-1 timeline actions.

Of course, the above example's class names ("MyMovieClip" and "MySound") should be customized with the names of the classes in your document. Add one class logging command per line within the actions panel, targeting every class that you've created within

your document. Use "addMovieClip()" to log MovieClip classes, and "addSound()" to log Sound classes. Whenever you add a new class to your library, be sure to log it within the main timeline's actions panel. If you ever rename or remove a class, also rename or remove its logging command accordingly. Also, each class should only be logged only once (otherwise you could create a naming conflict that results in an error).

Note to advanced Flash users: if you choose to package your library classes, then you'll need to import your class paths or else declare all classes by their full DOM path, such as:

```
addMovieClip( com.package.MyMovieClip );
```

Step 6: Publish

The last step is simply to publish your FLA document (File > Publish). This will compile the FLA into a SWF movie that the Lassie engine can load and extract assets from it.

In a perfect world, publishing your library SWF would work every time without error. However, Flash's compiling operation does a check of your document's configuration and will throw errors if it finds any problems (such as: class naming conflicts, missing references, etc). You'll know you've encountered an error if the Flash output window pops open with an error message while publishing. If you encounter a compiler error, don't panic. See the later section in this tutorial that covers common errors and troubleshooting. You'll need to address the error and republish the movie before the library is ready for use within Lassie.

After publishing a library SWF, upload the library into the “lib” directory of your project’s Shepherd development site. If the library is a voice library, upload it into “lib/voice”.

Errors and Troubleshooting

Compiler Errors

When you publish a FLA document, Flash "compiles" it into a SWF movie. The compiling process organizes and converts document assets into finished, usable media for the Flash player (and Lassie). However, the Flash compiler may encounter configuration errors while running, at which time it aborts the compile process and throws an error. Errors will appear in the Flash output window (which will automatically pop-open when an error is encountered). Compiler errors must be resolved before the document can publish and its resulting SWF can be used by Lassie.

So, first thing first: if you encounter a compiler error, don't panic. It's not the end of the world. The compiler exists because we are all human and we make mistakes. Compiler errors are designed to be our friend and help us work through those mistakes. And while the Flash compiler can be annoyingly picky, it is also extremely helpful with how specific it is about errors.

Finding the Source of the Problem

It is important to be aware that any custom ActionScript that you write into the Flash media within your library is also subject to the rigorous scrutiny of the Flash compiler. So, if you've followed this tutorial to the letter and still get compiler errors, there's a very good chance that it's being cause by your own scripts within the library. At which time - good luck. You're on your own.

To avoid uncertainty surrounding the source of an error, it's a good idea to develop your own custom scripts in a separate FLA document where you can publish and test them on their own. Once your own scripted media is working and error-free, move it over into your Lassie library document. This will isolate your own script errors from ones that may be encountered while setting up your library document.

Common Library Configuration Compiler Errors

Assuming that you can rule out your own scripts as a source of compiler errors, then there are only a couple errors that you are likely to encounter while following this tutorial. They include:

```
1120: Access of undefined property [name].
```

Check your class logging on frame-1. The compiler is telling you that it can't find a class that you've referenced, which probably means that you've mistyped a class name in the list, or else have deleted or declassified the library symbol. Look at the [name] portion of the error message to see the specific class name that Flash was not able to find.

```
1151: A conflict exists with definition [name] in namespace
internal.
```

Check your class logging on frame-1. This message probably means that you've declared the same class multiple times in the list. See the [name] portion of the message for reference as to which class name is in question.

Troubleshooting

You may encounter situations when your library SWF successfully compiles without errors, but not all of its class assets are available in the Lassie engine. Or in a worst-case scenarios, the library may be completely unavailable within Lassie. Don't panic. If the library compiled, then the error is most likely just a simple configuration issue. The following covers common configuration issues.

- *I updated my library FLA, but I'm not seeing any of the changes within Lassie.*

  Did you compile the updated FLA into a SWF? (File > Publish). If so, has the new SWF replaced the old SWF library at the location Lassie is reading it from... ie: have you uploaded the new SWF movie to your Lassie development site? If so, have you emptied your browser's cache and refreshed the page?

- *I've updated my library SWF and am successfully seeing some of the new/modified assets. However, some assets that I added are missing.*

  Look over your list of class declarations in the library's frame-1 actions. Did you forget to log the class? A classified asset will not be available within Lassie until it has been logged within frame-1 actions.

- *My library is doing nothing. It doesn't show up in the Lassie libraries list or it displays without any assets.*

  Check your document class path (Properties Panel > Document Class). Does your document class match what is specified in step-2 of this tutorial? If so, click the little pencil button next to the Document Class field. If a script file pops open, that's good (just close the script without editing). If an alert pops open, however, it probably contains the message: "A definition for the document class could not be found...". If you encounter this message, it means that your document is not linked correctly to the Library script file. Review steps 1 and 2 of this tutorial on how to correctly organize your production files so that all links work.