Using Class Based Development

In this unit, you will learn how to develop a Flash Lite application using class based development.

Adobe

Objectives

After completing this unit, you should be able to:

- Create ActionScript classes
- Link ActionScript classes to a visual `MovieClip` symbol
- Build properties and methods of an ActionScript class
- Managing scope issues in an ActionScript class using `Delegate`

Creating Classes in ActionScript

While Flash Lite ActionScript 2.0 includes many classes of objects, such as the `MovieClip` or the `Color` class, there will be times when you need to construct your own classes so you can instantiate objects based on a particular set of properties or methods.

- Building custom classes results in more modular, reusable code.

To create a class to serve as the blueprint for new objects, you must create a separate ActionScript class file. In ActionScript 2.0:

- All class definitions must be located in separate AS files. It is not possible to define classes on a Timeline (such as in Frame 1 of an Actions layer).
- The ActionScript (AS) file must have the same name as the class
- Filenames are case sensitive. By common convention, class names should begin with an upper case letter.

Creating Classes in ActionScript

Once the empty AS file has been created, you begin a class file as follows:

1. Define your class - a blueprint for objects you wish to use in your application - using the `class` keyword:

```
class ClassName
{

}
```

2. Declare properties (variables) to describe the characteristics of the objects to be created. (Scope attributes are discussed later):

```
class ClassName
{
 scope var propertyName:DataType;
}
```

Creating Classes in ActionScript

3. Declare a constructor function - a function which runs automatically to set up each object's initial state - for your class. A constructor must be named the same as the class itself (and its file), and must not have a return type:

```
class ClassName
{
 scope var propertyName:DataType;
 function ClassName(argument:DataType)
 {
   propertyName = argument;
 }
```

Creating Classes in ActionScript

4. Declare methods (functions) to define the behaviors of the objects to be created based on your class definition. (The possible attributes of a method are discussed later):

```
class ClassName
{
 scope var propertyName:DataType;
 function ClassName(argumentName:DataType)
 {
   propertyName = argumentName;
 }
 scope function methodName:ReturnType
 {
   return expression;
 }
}
```

Linking a Class to a MovieClip symbol

- Often in ActionScript you may use class files to describe and encapsulate data to be used on the mobile device.
- It can also be useful to link a class file to a visual object that you created in the Flash authoring tool, to enhance its reusability.
- In this unit, you will learn how to create a class file that has a visual element on the `Stage`. The visual element will be defined by a `MovieClip` symbol.

The steps for associating a class with a symbol are:

1. Create the `MovieClip` symbol and place it in the Library.
2. Build the class file.
3. In the symbol's Linkage Properties dialog, click Export for ActionScript and specify the fully qualified class name to be associated with this symbol.

Linking a Class to a MovieClip symbol

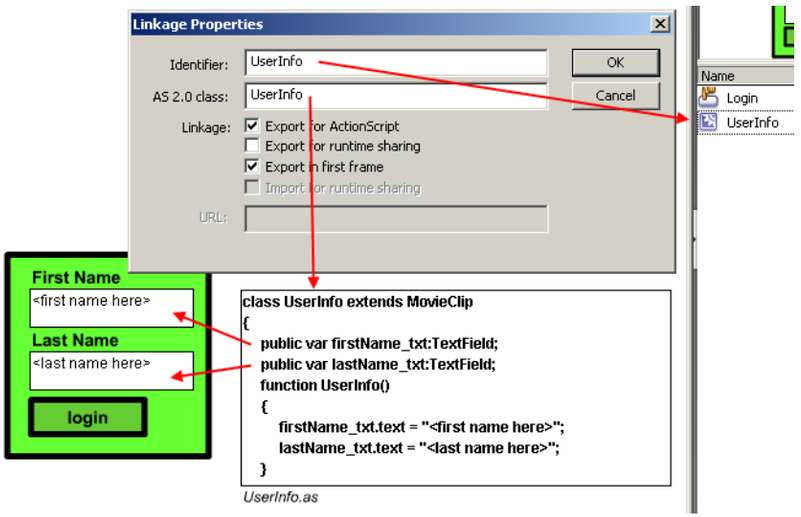


Figure 1: Linking a MovieClip to an ActionScript class

Note: While the Library name of a Symbol and its Linkage Identifier can be different, the best p ractice is to leave them the same (as suggested by default in the Linkage dialog).

Adobe

Extending the MovieClip symbol API

- The `MovieClip` class is the fundamental building block for visual content in the Flash Lite player.
  - All custom classes with a visual element should extend the `MovieClip` class, to inherit its API: all of its built-in properties, methods, and events.
  - To implement inheritance in ActionScript, you use the `extends` keyword. The general format for extending another class is:

```
class SubClass extends SuperClass
{
 // class definition
}
```

  - where the `SubClass` relates to the symbol to which your class file will be linked and the `SuperClass` is, generally, `MovieClip`.

Once this is done, all the properties, methods, and events of the `MovieClip` class may be directly referenced in your class, to manipulate and respond to changes to named visual elements (`Button`, `TextField`, and `MovieClip` objects) nested within the symbol linked to this class.

Setting classpaths

The classpath is a directory, or series of directories, where the compiler will look for class files when compiling an application. By default, the compiler will look in the following locations:

- Same folder as the FLA file
- Search - in order - directories specified as part of the global and document-specific classpath

The first class found of any given name will be used by the compiler.

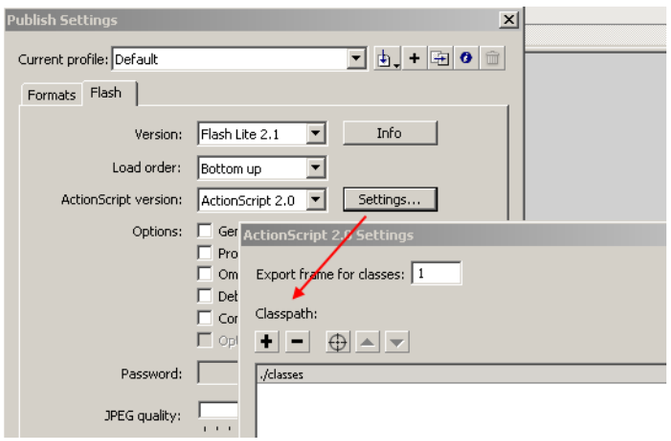
Modifying global classpaths

To modify global classpaths - used by all documents during compiliation - in the Flash 8 authoring tool, do the following:

1. Select File > Publish Settings to open the Publish Settings dialog box.
2. Click the Flash tab.
3. Verify that ActionScript 2.0 is selected in the ActionScript Version pop-up menu and click Settings.

Modifying global classpaths

4.n the ActionScript Settings dialog box, specify the frame from which the class definition should be made available (generally Frame 1).



Modifying global class path settings

To add or modify classpath directories, you may press

- ►Browse to Path (target symbol) button, browse to the folder you want to add, and click OK, or

Adobe

Modifying global classpaths

- Add New Path (+) button to add a new line to the Classpath list. Double-click the new line, type an absolute path, or a path to be evaluated relative to the position of any given FLA, and click OK.
- You may also select a directory, and press Remove (-), or use the arrow buttons to change the search order of the classpath directories.

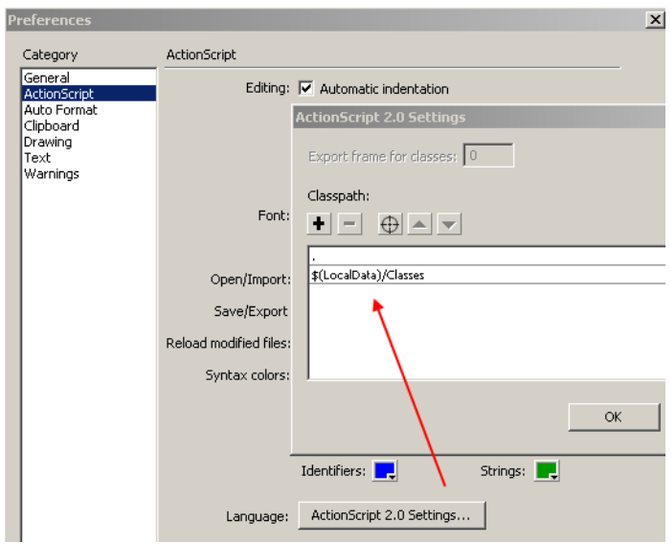
Modifying document-specific classpaths

Like-named classes within document-specific classpaths take precedence over classes within global classpaths. To modify document-specific classpaths - used solely by this document during compiliation do the following:

1. Select Edit > Preferences > ActionScript > ActionScript 2.0 Settings, to open the Publish Settings dialog box.

Modifying document-specific classpaths

2. Add, remove, and change the order of classpath directories in the same way as with global classpaths.



Modifying document-specific class paths

Notice that each document has two document-specific directories automatically with its classpath:

`.` `(dot operator)`: the same directory as the FLA itself

Adobe

Understanding Packages

A package is a group of related classes in one system directory with the classpath. Packages are created to help keep code logically organized.

In ActionScript 2.0, the package name is defined, using the dot operator, as part of the class name. Each prefix within the package name equates to a directory located beneath a classpath directory.

For example, `RestaurantMenu.fla` could be located in:

```
{docs}/clients/CafeTownsend/
```

Understanding Packages

Because the current directory of the FLA is, by default, part of the classpath, the following class could be written in the following directory, rendering it part of the package of classes defined in its name:

```
{docs}/clients/CafeTownsend/food/maindish/Pasta.as

class food.maindish.Pasta
{
 var dishName:String
 // no package prefix on the constructor
 function Pasta(dn:String)
 {
  dishName = dn;
 }
}
```

The `food/maindish/Pasta.as` directory and file could also be located under any directory which is part of the global or document-specific classpath.

Importing and using a packaged class

When a class resides in a package, it must be imported before it is used. The import keyword is used to identify, for the compiler (the software which "publishes" the SWF), where to find the code for the class.

Importing and using a packaged class

Here, the `Pasta` class must be imported to be used by the `Person` class. Without the import, the compiler wouldn't run, as it would not be able to locate the `Pasta` class.

```
{docs}/clients/CafeTownsend/Person.as

import food.maindish.Pasta;
class Person
{
 var food:Pasta;

 function eat()
 {
  food = new Pasta("Linguini");
 }
}
```

Note:

Walkthrough 1: Creating and Using ActionScript Classes

In this walkthrough, you will perform the following tasks:

- Create a new ActionScript class
- Add a `trace()` to the class consructor
- Link the class to a visual object

Defining class properties and methods

- A class is a blueprint of how to build an object.
- In simple terms, an object is a variable to which other variables are attached, describing its characteristics (properties), and functions are attached enabling its behaviors (methods).

Building and using methods

A function may be declared directly within a class. Commonly, they will be declared below the constructor.

```
class food.maindish.Pasta
{
 var dishName:String

 function Pasta(dn:String)
 {
  dishName = dn;
 }
 function getName():String
 {
  return dishName;
 }
 function setName(dn:String):Void
 {
  dishName = dn;
```

Building and using methods

- Whenever a function is defined within a class, it is known as a "method" of that class.
- Every time an instance of the `Pasta` class is instantiated, the resulting object will have functions attached to it - "methods"- called `getName()` and `setName()` which will write to and retrieve values from the `dishName` property, a variable which will also be attached to each instance.

The following code instantiates a new `Pasta` object and invokes its methods:

```
import food.maindish.Pasta;

var meal:Pasta = new Pasta("Lasagna");
trace(meal.getName()); // displays "Lasagna"
meal.setName("Spaghetti");
trace(meal.getName()); // displays "Spaghetti"
```

Method and property attributes

Three attributes may be used to control property or method access.

Keyword	Meaning
`public`	The method or property can be used outside the class where created.
`private`	The method or property can be used only within the class, or a class which inherits from this class.
`static`	The method or property cannot be used through an instance of this class, only through a reference to the class name itself

`public` access is the default in ActionScript 2.0 if no other access attribute is specified. It is the best practice to explicitly specify an attribute, such as public or private, for a method or property.

Method and property attributes

Consider the following code example:

```
class food.maindish.Pasta
{
 private var dishName:String
 function Pasta(dn:String)
 {
  dishName = dn;
 }
 public function getName():String
 {
  return dishName;
 }
 public function setName(dn:String):Void
 {
  dishName = dn;
 }
}
```

\Static Methods

When using `static` methods, the method must be invoked through a direct reference to the class name. A `static` method cannot be invoked through an instance of the class in which it is defined.

A class with a static method is shown here:

```
class Product
{
 function Product() {} //constructor
 static function getType():String
 {
  return "Product";
 }
}
```

In order to call the `static` method, you directly refer the name of the class. You do not create an instance of the class.

```
trace(Product.getType()); // displays "Product"
```

\Static Methods

Other important points about `static` methods and/or properties are:

- Only other `static` properties or methods are used in `static` methods.
- The `static` attribute can be used only within a `class` definition.

Static methods are used throughout the Flash Lite API. For example, `Math.random()` and `Delegate.create()`.

Walkthrough 2: Adding Methods and Properties to a Class

In this walkthrough, you will perform the following tasks:

- ▶Create class properties
- ▶Create class methods

Managing Scope issues in class files

- The term "scope" refers to the context in which a particular statement or code block executes.
- All identifiers (names) must be unique within each scope. The context is defined by the current target of the keyword `this`.

Understanding the problem

- When an event handler is directly assigned to an object's event, the context - the target of `this` - will be the object to which the assigned event belongs, rather than the object in which the assignment was made.
- If the class creates nested objects within itself then assigns event handlers to them this may create problems when writing code within a class file,

Understanding the problem, code example

```
class BigBall extends MovieClip
{
 private var smallBall:MovieClip;
 function BigBall()
 {
  smallBall = attachMovie("SmallBall", "smallBall",
  getNextHighestDepth());

  // directly assign event handler
  smallBall.onPress = doPress;
 }
 private function doPress():Void
 {
  trace(this);
  // displays _level0.bigBall.smallBall
  // so 'scope' here is smallBall, not bigBall
 }
}
```

Understanding the problem

This may create problems if the event handler function needs to execute other methods of the enclosing `BigBall` class, as some form of relative pathing - such as using the `_parent` keyword - would be required, resulting in code which is more difficult to maintain.

Changing scope using the Delegate class

- The static `create()` method of the `Delegate` class allows you to define the specific scope in which an event handler will execute.

- You must `import` the `Delegate` class to use it.

Changing scope using the Delegate class

```
import mx.utils.Delegate;
class BigBall extends MovieClip
{
 private var smallBall:MovieClip;
 function BigBall()
 {
  smallBall = attachMovie("SmallBall", "smallBall",
  getNextHighestDepth());

  // delegate current scope to event handler
  smallBall.onPress =
   Delegate.create(this, doPress);
 }
 private function doPress():Void
 {
  trace(this);
  // displays _level0.bigBall
  // so 'scope' is now bigBall, not smallBall
```

Changing scope using the Delegate class

- The benefit of delegating an event handler to an event, rather than directly assigning one, is that a nested object's event handler - `doPress()`
- In this example - can directly invoke other methods of the enclosing class, without relying on potentially fragile relatively pathed (`_parent`) code. This generally results in easier long-term code maintenance.

Walkthrough 3: Using the Delegate Class

In this walkthrough, you will perform the following tasks:

- Trace the scope that the `onRelease` event executes in.
- Import the `Delegate` class
- Delegate the `onRelease` event to the `MessageScreen` class

Summary

- An object is an instance of a class.
- A `class` is defined in a text file, and is comprised of `var` (property) and `function` (method) declarations.
- A constructor function runs automatically as each instance is created, to initialize its property values.
- The class name, file name, and constructor name match exactly, and conventionally begin with an upper-case letter.
- Classes may be logically organized into packages, relating to the directories in which the class files reside.
- The classpath is the directory or list of directories Flash 8 is told to look in to find class definitions.
- Properties and methods - other than the constructor function - may be `public` or `private`.

Summary

- Public properties and methods are accessible to other code outside the class, private properties and methods are not.
- A `class` may be linked to a `MovieClip` or `Button` symbol, and its code can interact with named objects within the symbol.
- A `static` property or method is shared by all objects of that class, and is referred to through the class name, not an instance.
- The path keyword `this` refers to the last referenced object.
- Where objects are created within a class, and event handlers assigned, the class context may be passed to a child object's event handler using the static `create()` method of the `Delegate` class.