Recently a new breed of developers has taken on the challenge of creating applications that enable users to create their own content. In this chapter, you'll build one such application—an online Movie Maker that enables users to build short movies from a set of premade animations.

Working on projects like Movie Maker presents a special challenge to developers who are learning ActionScript. This project takes advantage of some powerful new features, including movie clip buttons, dynamic event handlers, and the `registerClass` method.

11

MOVIE MAKER

by Jason Krogh
with original audio and design by Brian Ziffer and James Lloyd of **www.systemsoular.com**

Difficulty rating: ★★★★★

Made with:

Animation created using Adobe Photoshop, Extreme 3D, LightWave, AfterEffects, ImageReady
Audio processed using Reason, Max/Nato Custom applications
Original audio and design by Brian Ziffer and James Lloyd, **www.systemsoular.com**

Final file size: 30KB (**moviemaker.swf**), 2MB (**assets_ziffer.swf**)

Modem download time: 7 seconds + time for assets

Development time: 5 days

IT WORKS LIKE THIS

The Movie Maker enables users to create a movie by arranging animated clips within a timeline. The drag-and-drop interface is designed to be easy to use while giving users a great deal of flexibility.

The Movie Maker timeline contains 480 frames and playback is set at 20 frames per second (fps). To create a movie, the user drags a timeline clip from a palette into one of three tracks. These timeline clips are visual representations that indicate when the animated clips begin and end as well as where they sit in the stacking order. (For example, items in track 1 appear on top of items in track 3.)

The movie displays in a preview window. This preview is constantly updated based on the arrangement of the timeline clips and the position of the playhead marker. The user can use the playback controls to play, stop, rewind, or fast forward through the movie.

Note: Flash MX makes it easier to take advantage of *object-oriented programming* (OOP) techniques. An *object* is just a set of shared properties (information) and methods (capabilities). The example most Flash users are already familiar with is the `MovieClip` object. All movie clips share certain properties, such as `_x`, `_y`, and `_alpha`, as well as methods, such as `getDepth` and `loadMovie`. Although a full discussion of OOP concepts is beyond the scope of this chapter, a few relevant topics are covered.

PREPARING TO WORK

Before you get started, you need to copy the **11_Movie_Maker** folder onto your hard drive and launch Flash MX. Take a moment to try out the finished Movie Maker, and then open the starter source file.

1. From your hard drive, open the **moviemaker.html** page in your web browser. Drag timeline clips from the palette into one of the three tracks and experiment with the playback controls.
2. Open the **mm_start.fla** file from the **11_Movie_Maker** folder on your hard drive.

   The start file includes all the buttons, movie clips, and some of the functions needed to make things work.
3. Advance to frame 2 and examine the layer structure in the timeline.

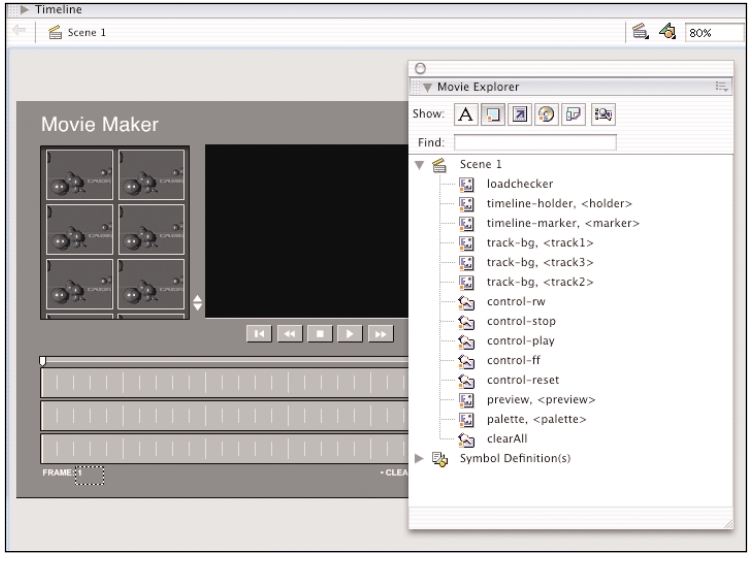
Note: This project uses two Flash files. The main file (**moviemaker.swf**) contains the interface and programming. The assets file (**assets_ziffer.swf**) contains the animation and a movie clip containing the background audio. By having the assets in a separate movie, you can more easily manage the artwork without interfering with the core of the application. The assets file contains a set of movie clips with linkage identifiers, enabling you to use the `attachMovie` method to place them on the stage programmatically. Examine the **assets_sample.fla** file to see how it is organized.

4 Open the Movie Explorer panel from the Window menu and examine the movie clips and buttons.

The interface on the main timeline is set up with four parts: the palette, the timeline area, the preview window, and the playback controls. Nearly all the code is found on the main timeline.

Note: The start file already includes most of the functions needed for the Movie Maker. These functions are located on the first keyframe of the **Functions** layer. Each function performs a certain part of the work and many correspond to the actions of the user.

5 Open the Library from the Window menu. Open the **Track Clip** folder in the Library and examine the track-clip symbol.

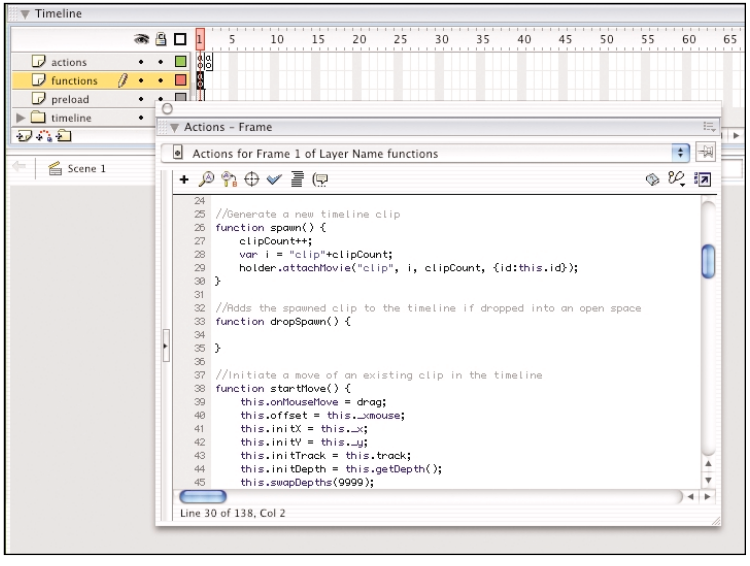


The Movie Explorer shows the movie clips and buttons that have been placed on the stage.

Creating New Timeline Clips

The timeline clips are the central element of the Movie Maker. The number, type, and position of these clips represent the composition. Timeline clips are created by the `spawn` function, which is triggered when the user clicks an item in the palette.

1 On the main timeline, with the Actions panel open, select the keyframe in frame 1 of the **Functions** layer. Locate the `spawn` function in the Actions panel.



The `spawn` function is used to create timeline clips.

2 Add this code directly after the line defining the `spawn` function.

```
clipCount++;
var i = "clip"+clipCount;
holder.attachMovie("clip", i, clipCount, {id:this.id});
```

This code attaches a new instance of the timeline clip symbol to the **holder** movie clip. You use four arguments with the `.attachMovie` method: the linkage identifier `"clip"`, and then the instance name, depth value, and an `initObject`.

The code for the `initObject` argument makes use of an abbreviated syntax for creating objects. The `{id:this.id}` statement creates a simple object with a single property id.

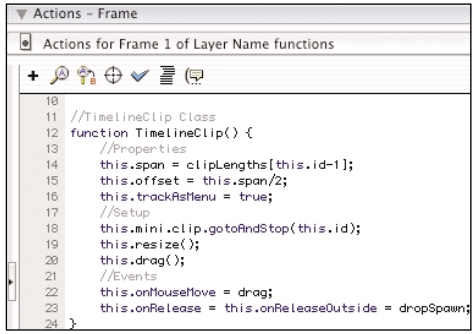
Macromedia Flash MX: The optional `initObject` argument is a new feature of Flash MX. It enables you to assign a set of methods and properties to the clip as it is attached. In this case, you use this feature to pass the `id` to the newly created movie clip instance. You will see how this is used when you look at the `TimelineClip` class.

Defining the TimelineClip Class

A *class* is used to define a set of objects with common features (for instance, a class of trees or a class of kindergarten kids). In ActionScript you define a class using a function. The timeline clips added by the `spawn` function share certain properties (for instance, which animation segment they represent) and methods (for example, resizing themselves). Using a class enables you to define a single blueprint for your timeline clips.

Note: The distinction between a function and a method is subtle but important. A *method* is a function that has been associated with an object. The beauty of methods is that they have direct access to the properties of the object to which they are attached.

1 Locate the `TimelineClip` function in the Actions window. Add this code directly under the line marked `//Properties`.




```
this.span = clipLengths[this.id-1];
this.offset = this.span/2;
this.trackAsMenu = true;
```

The `id` used when setting the `span` property is passed to the object by the `attachMovie` method invoked in the `spawn` function. It identifies which segment of animation the timeline clip represents. (They are numbered 1–15.) The `span` represents the length of the clip in frames, and the `offset` is used when positioning the clip while it is being dragged.

2 Add this code directly under the line marked `//Setup`:

```
this.mini.clip.gotoAndStop(this.id);
this.resize();
this.drag();
```

These three lines are performed as soon as the new timeline clip is created. The `gotoAndStop` action changes the mini view of the animation based on the `id` of the clip. You then call the `resize` and `drag` methods right away. These two methods are defined by existing functions.

3 Add this code directly under the line marked `//Events`:

```
this.onMouseMove = drag;
this.onRelease = this.onReleaseOutside = dropSpawn;
```

Here you assign callback functions to the built-in movie clip event handlers. The first line triggers the `drag` function when the user moves her mouse, and the second line triggers the `dropSpawn` function when the user drops the clip.

4 Add this code directly under the line marked `//Register Class`:

```
TimelineClip.prototype = new MovieClip();
Object.registerClass("clip", TimelineClip);
```

The first line tells the player that your `TimelineClip` inherits all the built-in methods and properties of the `MovieClip` class. Any of the new methods and properties you attach are added to those already available in all movie clips.

The second line makes use of a powerful new ActionScript command. You use the `registerClass` method to associate the `TimelineClip` class with the Library symbol with the linkage identifier "`clip`". Note that you are not associating the class with an instance but rather the Library symbol. This means that all new instances of the symbol with the linkage identifier "`clip`" are members of the `TimelineClip` class and therefore have all the properties and methods defined for this class.

Adding a new timeline clip to the movie.

The spawn function attaches and initializes a new clip. The drag function moves the clip to follow the cursor. The dropSpawn function adds or removes the clip based on its position.

MacromediaFlash MX: In Flash 5, you used movie clip events by placing code within an onClipEvent (MovieClipEvent) statement. This approach severely limited your ability to centralize your code, because these statements needed to be placed directly on movie clip instances. In Flash MX you can assign callback functions to these events programmatically. Not only that, but there are also new handlers for button events (for instance, .onPress), which you can use with movie clips.

The ability to add button events to movie clips can simplify complex movies considerably. In Flash 5 it was often necessary to nest a button inside a movie clip. This was necessary any time you needed to catch button events while being able to manipulate visual properties of the button (such as, _x, _y, _width, and _alpha). Now we can add those events directly onto the movie clip instance. In fact, because of these new features there is very little reason for experienced developers to use button symbols.

5 Add this code directly under the line marked //Methods:

Because these methods are common to all members of the TimelineClass, you attach them to the class's prototype. By doing so, all objects created from this class do not contain their own copies of these methods but rather share the exact same set of code. The result is more efficient use of memory in the Flash player.

```
TimelineClip.prototype.dropLocationOkay = dropLocationOkay;
TimelineClip.prototype.updateQuePoints = updateQuePoints;
TimelineClip.prototype.drag = drag;
TimelineClip.prototype.resize = resize;
```

Assign functions to act as methods for the timeline clips.

Adding Drag and Drop Behavior

Now that you have your new timeline clip, you need to create some of the methods that it will use. Four methods are involved in this process: drag, dropSpawn, updateQuePoints, and dropLocationOkay. You are going to add the first two of these.

The drag method has been assigned to the onMouseMove event and is called continuously until you remove the handler. It positions the timeline clip according to the location of the cursor and provides user feedback as the clip is moved around. It does this with the help of updateQuePoints and dropLocationOkay. The dropSpawn method is called when the user releases the mouse button after having just created a new timeline clip.

1 Locate the `drag` function in the Actions panel. Add this code directly following the line defining the `drag` function:

```
this._x = this._parent._xmouse-this.offset;
this._y = this._parent._ymouse;
this.updateQuePoints();
```

Repositioning the timeline clip.

The first two lines reposition the clip based on the user's cursor position. The `offset` is used to handle situations where the user picks up the clip by the edge.

The `updateQuePoints` updates important information about each timeline clip as it is manipulated by the user. It sets three properties based on the clip's position: `begin`, `end`, and `track`. `begin` and `end` are set to the first and last frame where the clip is present. The `track` property is set to one of four values: `1`, `2`, `3`, or `null` if the clip is not in a track.

2 Add this code:

```
if (this.dropLocationOkay()) {
      this._y = _root["track"+this.track]._y;
      this._alpha = 100;
} else {
      this._alpha = 50;
}
updateAfterEvent();
```

Testing the location of clips.

The `dropLocationOkay` method uses the new `begin`, `end`, and `track` values and tests to see whether they overlap with any existing clips. The `dropLocationOkay` method returns `true` if the location is acceptable.

If the method returns false, we know that the timeline clip is not in a valid location. We indicate this to the user by setting the alpha property to 50%. Otherwise you set the `alpha` to `100%` and snap the clip to the nearest timeline.

3 Add this code directly following the line defining the `dropSpawn` function:

```
delete this.onMouseMove;
if (!this.dropLocationOkay()) this.removeMovieClip();
this.onPress = startMove;
this.onRelease = this.onReleaseOutside = dropMove;
```

Deleting the clip when its location is unaccepatble.

The first line removes the `onMouseMove` handler. This stops the `drag` method from being called. The next line checks to see whether the user released the clip in an acceptable position. If not, the clip is deleted (along with all of its methods and properties).

Note that when the user moves an existing clip things are treated a little differently. Because of this you have separate `move` and `dropMove` methods. The last line of code reassigns the `.onRelease` and `.onReleaseOutside` events to trigger the `dropMove` method the next time the clip is dropped.

The displayFrame Function

The playback of the movie is controlled by the `displayFrame` function. This function calculates the current frame and then goes through each timeline clip to see which clips are present in that frame. The flexibility in the user interface makes your work a little more difficult. You cannot assume that the movie is going to be played from start to finish. Instead you have to consider other scenarios, such as the following:

- The playhead is dragged back and forth.
- Playback is started while midway through a timeline clip.
- The `playSpeed` is set for rewind or fast forward.
- The timeline clips are being moved during playback.

In short the function must work in isolation without knowing what has happened before or after it. The user can move the clips or the playhead and always see (and hear) the correct animation in the preview display.

1 Locate the `displayFrame` function in the Actions panel. Add this code to the `displayFrame` function:

```
var occupied = new Array(3);
if (playing) moveMarker();
frame = 1+Math.floor((marker._x-tl_left));
```

Tracking the presence of timeline clips.

The first line declares a new array called `occupied`. You will use this later to monitor which tracks contain clips on the current frame.

The next line calls the `moveMarker` function if the movie is playing. The `moveMarker` changes the position of the playhead based on the `playSpeed` variable. Next you calculate the current frame based on the playhead marker's position.

2 Add this code directly following the statements added in Step 1:

The first line sets up a `for-in` loop. This type of loop is used to step through the contents of an object. In this case, you are checking each item inside the holder movie clip.

The body of the `for-in` loop checks each timeline clip within the holder to see whether the current frame lies between the clip's `begin` and `end` values. If it is not already present, the clip is added using the `attachMovie` method.

If the movie is playing normally and the clip is not being dragged, you let the animation segment play. If the movie is not playing back or the clip is being moved, however, you instead set the frame position of the clip based on how far into the clip you are.

```
for (item in holder) {
if (holder[item]._alpha == 50) continue;
if (frame >= holder[item].begin and frame <= holder[item].end) {
      var instance = "i"+holder[item].track;
      if (preview[instance].id != holder[item].id) {
            var linkage = "clip"+holder[item].id;
            var depth = 4-holder[item].track;
            preview.attachMovie(linkage, instance, depth);
            preview[instance].id = holder[item].id;
      }
      occupied[holder[item].track] = true;
      var isDragging = (holder[item].onMouseMove == drag);
      if (playing && playSpeed == 1 && !isDragging) {
            preview[instance].play();
      } else {
            var clipFrame = 1+frame-holder[item].begin;
            preview[instance].gotoAndStop(clipFrame);
      }
}
}
```

3 Add this code directly after the end of the `for-in` loop:

First you remove any animation clips that are no longer needed.

Next you handle the playback of the background audio. The background audio is treated differently than the other elements. The background audio clip is attached in frame 2 of the Actions layer and is present throughout the movie.

If the movie is playing, the movie clip containing the audio is told to play. Otherwise you cue the movie clip to the right point so that when the movie is played, the audio starts in the appropriate place.

```
for (var i=1;i<4;i++) {
      if (!occupied[i]) preview["i"+i].removeMovieClip();
}
if (playing && playSpeed == 1) {
      preview.bgAudio.play();
} else {
      preview.bgAudio.gotoAndStop(frame);
}
```

4 Now test your movie. If it is not functioning as expected, make sure the assets file is located in the same directory. Use the finished file, **moviemaker.fla**, to check the code you have added.

Note: It is extremely rare that a programmer writes polished code in a single pass. This is especially true for routines such as `displayFrame`.

When tackling a complex problem, it's often best to start with a diagram and then translate it into pseudo-code. The descriptions of the code in this chapter are examples of pseudo-code, or explanations in plain English of what the code does. However, real-life pseudo-code more often consists of scribbled notes on the back of a piece of paper.

You can create the framework for your projects using a combination of function declarations, comments, and trace statements. Here is a simple example:

```
//This function returns the average of two integers.
function average(a,b) {
      trace("calculating an average");
}
```

Now you can choose where to start filling in the details. Much like taking a test, it's sometimes better to start with the easy questions and return to the tough ones later. For complex routines, you often need several passes to get the exact functionality you need. Getting comfortable with this iterative approach will help you tackle all sorts of complex programming challenges.

Summary

Flash MX enables developers to create sophisticated applications. However, building such applications involves more than a knowledge of ActionScript. It also requires the ability to step back, look carefully at the steps involved, and translate these steps into code.

In this chapter you looked at a few of the new features of Macromedia Flash MX that enable you to create flexible centralized code. Specifically, you made use of dynamically assigned event handlers and classes to define a blueprint for a set of objects.

Although creating applications requires a certain discipline in how you approach your work, don't ever be afraid to write, rewrite, and experiment as you go.