

Solutions and Examples for Flash MX Developers



ActionScript Cookbook

O'REILLY®

Joey Lott

ActionScript Cookbook

ActionScript Cookbook

Joey Lott

Storing Persistent Information

16.0 Introduction

You already know how to work with data that is stored in memory as long as a Flash movie is open. In fact, almost all data you work with in ActionScript is of this type. But what if you want to be able to store data between Flash movie sessions? Or what if you want to be able to share data between Flash movies running on the same client, or even on different clients? To accomplish these feats, we'll need to look at how we can store and transmit data outside of the Flash Player.

This chapter addresses both client-side and server-side data. First, we'll examine client-side data. Prior to Flash 6, the only way to store persistent data on the client was with complex techniques that invoked external JavaScript functions to read and write cookies. However, Flash 6 introduced *local shared objects* to ActionScript, making it possible to store and retrieve persistent data on the client without the use of these other, complicated techniques.

We'll also take a closer look at server-side data. Prior to Flash 6, server-side data manipulation was limited to invoking server-side scripts (using *loadVariables()* or an XML object) to read and write data to text files and/or databases on the server. The problem was, of course, that this functionality required that you had the necessary scripts on the server. While this kind of functionality still has its place (although the *LoadVars* class supercedes the older *loadVariables()* method), this chapter examines how *remote shared objects*, used in conjunction with FlashCom, allow us to work with server-side data in a different way.

Shared objects come in several varieties. Local shared objects (LSOs) are similar to browser cookies in that they are stored on the client's machine. Local shared objects are useful for storing the same kinds of information for which cookies have traditionally been used, such as the capability for a web site to remember a user so that the user does not have to manually log in during each visit. But LSOs are more powerful than cookies because they can store more data than cookies and store native ActionScript

datatypes. Local shared objects are available to any Flash 6 movie, and they do not require any additional software on the client or on the server.

In contrast with local shared objects, remote shared objects (RSOs) are available only for movies that are clients of a Flash Communication Server MX (FlashCom) application (see Chapter 14). Remote shared objects allow you to store persistent data that is accessible by all clients of that application. Remote shared objects can also be used for nonpersistent data sharing between clients (see Recipe 16.5).

Local shared objects exist on the client only. On the other hand, remote shared objects must be opened on both the client and the server. Throughout this chapter, we refer to shared objects that are opened on the client (whether local or remote shared objects) as *client-side shared objects*, and we refer to shared objects opened on the server (using code within an *.asc* file) as *server-side shared objects*. This is an important distinction because the ActionScript that you use in each case is different.

16.1 Storing and Retrieving Locally Persistent Information

Problem

You want to store information on a client's computer that persists between movies and sessions.

Solution

Use a local shared object (LSO).

Discussion

Local shared objects are to Flash what cookies are to web browsers—but more so. They are called “super cookies” by some developers because they allow you to store potentially large amounts of data (if the user allows) and because they also allow you to store and retrieve many native ActionScript datatypes (as well as objects created from custom classes). Local shared object files are saved to the client computer. Flash movies within the same domain can write to and read from these files by means of ActionScript's *SharedObject* class.

The static *getLocal()* method is the mechanism by which LSOs are both created and opened for reading. The method requires at least one parameter—a string that specifies the name of the shared object to create or open:

```
my_l_so = SharedObject.getLocal("myFirstLSO");
```

The *getLocal()* method attempts to first locate an existing LSO by the specified name stored on the client computer. If none is found, Flash creates a new LSO with that

name. In either case, the existing or new LSO is opened. The `getLocal()` method returns a *SharedObject* instance. It is used instead of the *new* operator to instantiate a new local shared object.

For a complete example of how to create, write to, and save a local shared object, see Recipe 16.4.

See Also

Recipe 16.5 discusses sharing data between movies. Also refer to Recipe 2.8 for details on opening the Flash Player Settings dialog box to the Local Storage tab. See Recipe 16.2 for important information on how to store data in a local shared object. See Recipe 16.3 for important information on how to retrieve data from a local shared object. See Recipe 16.4 for information on the *SharedObject.flush()* method, which is used to manually save data to a shared object.

16.2 Adding Data to a Client-Side Shared Object

Problem

You want to add data to a client-side shared object (whether it be a remote or local shared object).

Solution

Add the values as properties of the shared object's data object.

Discussion

Shared objects have a special built-in property named `data`. The `data` property is an object to which you should add any information that you want to save to the shared object:

```
// Store a username value to the shared object.  
mySO.data.username = "Joey";
```

Properties attached directly to the shared object, rather than to its `data` property, are not written to the shared object. Thus, attaching properties directly to the shared object is useful only for data that does not need to persist between sessions. Therefore, the following code is potentially an erroneous attempt to save data:

```
mySO.someVal = "this value is lost when the movie closes";
```

The correct approach is to attach the value to the `data` property, as follows:

```
mySO.data.someVal = "this value is stored in the shared object";
```

You can store several native ActionScript datatypes to the shared object's data property, as follows.

```
mySO.data.myArray = new Array("a", "b", "c");  
mySO.data.myDate = new Date();
```

However, you cannot store movie clips, buttons, text fields, or shared objects themselves to a shared object's data property such that the data persists correctly between sessions.

You can also store objects created from custom classes to a shared object's data property. However, for the object to be properly cast (interpreted as the intended datatype) when it is retrieved in the next session, you must use the *Object.registerClass()* method. If you do not register your custom class, ActionScript won't know what to do with the data retrieved from the shared object.

The conventional use of the *registerClass()* method is to associate a class to a symbol in the Library. You should do this when you want to develop components with class definitions. However, you can also use the *registerClass()* method to register a class to an arbitrary identifier:

```
function MyClass () {}  
  
Object.registerClass("MyClassID", MyClass);
```

This code tricks the object into storing information about the class that created it. Therefore, when the data is retrieved from the shared object, Flash knows what kind of object it is.



The custom class must be defined and registered in all movies that retrieve the custom object data from a shared object.

There is one additional consideration you must address to successfully recast custom datatypes when they are retrieved from a shared object. When an object is retrieved from a shared object, all the properties are set and then the constructor for the class is called. This process will likely wipe out some, if not all, of the properties of the object if your constructor includes any initialization functionality. To understand this potential problem, consider the following example:

```
// Define a custom class. The constructor sets a property in the new object.  
function MyClass(val) {  
    this.val = val;  
}  
  
// Register the class.  
Object.registerClass("MyClassID", MyClass);  
  
// Create or open a local shared object.  
my_l_so = SharedObject.getLocal("myFirstLSO");
```

```

// If the shared object doesn't already have an objA object, create a new MyClass
// object and assign it to the shared object. Otherwise, if objA already exists
// (meaning that it was saved from a previous session), display the value of its val
// property in the Output window.
if (my_l_so.data.objA == undefined) {
    my_l_so.data.objA = new MyClass("yay");
    my_l_so.flush();
} else {
    trace(my_l_so.data.objA.val);
}

```

In this example, the value of the object’s `val` property is always undefined. The reason is that even though the object was initially saved with the value “yay”, the constructor is called with an undefined value each time the shared object data is retrieved. You can solve this problem by enclosing all the initialization code within the constructor in an *if* statement, as follows, which prevents the property from being erroneously reinitialized:

```

function MyClass(val) {
    if (!this.inited) {
        this.inited = true;
        this.val = val;
    }
}

```

When you change the constructor in this way, the original value for `val` is retained. When the *MyClass* object is first created, the `inited` property does not exist. Therefore, the *if* statement’s condition is true. However, when the object is retrieved from the shared object, the `inited` property is true, and the initialization code does not execute. In this way the `val` property is not overwritten.

See Also

Recipes 16.3, 16.10, and 17.2

16.3 Reading Values from a Client-Side Shared Object

Problem

You want to read values from a client-side shared object (either a remote or local shared object).

Solution

Read the values from the properties of the shared object’s `data` property.

Discussion

There is nothing difficult about reading the values from a client-side shared object. All persistent values are stored in the shared object's data property, so you simply read the values from the data property, as follows:

```
// Read the value of myProperty from the shared object, mySO, and write it to the
// Output window.
trace(mySO.data.myProperty);
```

See Also

Recipe 16.2. Also refer to Recipe 16.9 for important differences when reading data from a remote shared object.

16.4 Saving a Local Shared Object

Problem

You want to save local shared object data to the client computer.

Solution

Use the *SharedObject.flush()* method in the Flash movie.

Discussion

Flash automatically attempts to save local shared object data to disk when the movie is unloaded from the Player (such as when the Player closes). However, it is not a good practice to rely on the automatic save functionality, as there are several reasons why the data might not save successfully. Instead, you should explicitly instruct the local shared object to write the data to disk using the *SharedObject.flush()* method:

```
flushResult = my_l_so.flush();
```

When the *flush()* method is invoked, it attempts to write the data to the client computer. The result of a *flush()* invocation can be one of three possibilities:

- If the user set the local storage for the domain to “Never”, the data is not saved and the method returns false.
- If the amount of disk space required to save the local shared object's data is less than the local storage setting for the domain, the data is written to disk and the method returns true.
- If the user has not allotted as much space as the shared object data requires, he is prompted to allow enough space or to deny access to save the data. When this happens, the method returns “pending”. If the user chooses to grant access, the extra space is automatically allotted and the data is saved.

In the third case, in which the *flush()* method returns “pending”, there is an additional step you can take to determine whether the user grants or denies access to save the data. When the user makes a selection from the automatic prompt, the *onStatus()* method of the shared object is automatically invoked. It is up to you to define the method to handle the results in the way that is appropriate for your application. When the callback method is invoked, it is passed a parameter. The parameter is an object with a code property that is set to “SharedObject.Flush.Success” if the user granted access or “SharedObject.Flush.Failed” if the user denied access.

Here is an example that invokes *flush()* to save the data explicitly and then handles the possible responses:

```
my_l_so = SharedObject.getLocal("myFirstLSO");
my_l_so.data.val = "a value";
result = my_l_so.flush();
// If the flush operation completes, check the result.
// If the operation is pending, the onStatus() method of the
// shared object is invoked before any result is returned.
if (result) {
    // Saved successfully. Place any code here that you want to execute after the data
    // was successfully saved.
}
else if (!result) {
    // This means the user has the local storage settings to 'Never.' If it is
    // important to save your data, you may want to alert the user here. Also, if you
    // want to make it easy for the user to change his settings, you can open the local
    // storage tab of the Player Settings dialog box with the following code:
    // System.showSettings(1);
}

// Define the onStatus() method for the shared object.
// It is invoked automatically after the user makes a selection
// from the prompt that occurs when flush() returns "pending."
my_l_so.onStatus = function (infoObj) {
    if (infoObj.code == "SharedObject.Flush.Success") {
        // If the infoObj.code property is "SharedObject.Flush.Success", it means the
        // user granted access. Place any code here that you want to execute when the
        // user grants access.
    } else if (infoObj.code == "SharedObject.Flush.Failed") {
        // If the infoObj.code property is "SharedObject.Flush.Failed", it means the user
        // denied access. Place any code here that you want to execute when the user
        // denies access.
    }
};
```

If you know in advance that a shared object is likely to continue to increase in size with each session, it is prudent to request a larger amount of local storage space when the shared object is created. Otherwise, each time the current allotted space is exceeded, the user is prompted again to accept or deny the storage request. Setting aside extra space avoids repeatedly asking the user for permission to store incrementally more

data. You can request a specific amount of space when you call the *flush()* method by passing it a number of bytes to set aside for the shared object:

```
// Request 500 KB of space for the shared object.  
result = mySO.flush(1024 * 500);
```

16.5 Sharing Information Between Movies Within the Same Domain

Problem

You want two movies within the same domain to have access to the same local shared object.

Solution

Specify a local path parameter when creating and opening the local shared object.

Discussion

By default, local shared objects are saved to a path on the client computer that is unique to the domain, path, and name of the *.swf* file that is calling the *getLocal()* method. This prevents name conflicts between local shared objects from different domains or even different movies on the same domain. For example, on a system running Windows XP, if a movie named *myMovie.swf* served from *www.person13.com/ascb/* writes a local shared object named *myFirstLSO*, the data is saved to the following location:

```
D:\Documents and Settings\[UserName]\Application Data\Macromedia\Flash  
Player\person13.com\ascb\myMovie.swf\myFirstLSO.sol
```

The name of the *.swf* file is included in the path to which the LSO is saved so that it will not conflict with an LSO named *myFirstLSO* created by another movie served from the same domain and path. However, in some cases, you want two movies on the same domain to have access to the same LSO. In these cases, you should use the optional local path parameter when creating and opening the LSO using *getLocal()*.

The local path parameter (the second parameter passed to *getLocal()*) must be a string that specifies the full or partial path to the *.swf* file that created the LSO. For example:

```
my_l_so = SharedObject.getLocal("myFirstLSO", "/");
```

If the preceding code exists in *myMovie.swf*, which is served from *www.person13.com/ascb/*, the local shared object is stored at the following location:

```
D:\Documents and Settings\[UserName]\Application Data\Macromedia\Flash  
Player\person13.com\myFirstLSO.sol
```

An LSO created in this way can be opened by any other Flash movie in the same domain with the following line of ActionScript:

```
my_l_so = SharedObject.getLocal("myFirstLSO", "/");
```

It is important to understand that a movie can only create and/or open an LSO within the same full or partial path. To understand this, consider an example with two Flash movies: *movieOne.swf* and *movieTwo.swf*. Both movies are served from the same domain—*www.person13.com*—but at different paths. *movieOne.swf* is served from *www.person13.com/ascb/firstGroup/*, and *movieTwo.swf* is served from *www.person13.com/ascb/secondGroup/*. In this scenario, *movieOne.swf* can create and read LSOs with any of the following local path values:

```
/
/ascb
/ascb/firstGroup
```

and *movieTwo.swf* can create and read LSOs with any of the following local path values:

```
/
/ascb
/ascb/secondGroup
```

Therefore, if you want both movies to be able to access a common LSO, you must specify one of the two local paths that the movies share (*/* or */ascb*) when you invoke the *getLocal()* method.

To illustrate how you can share data between two (or more) Flash movies within the same domain, let's take a look at an example. If the movies don't exist within the same directory, we must specify a local path that is common to both of them in the directory hierarchy. Let's start by looking at what happens if we fail to specify a common local path:

1. Create a new Flash document, and on the first frame of the main timeline add the following code:

```
this.createTextField("message_txt", 1, 100, 100, 100, 20);
message_l_so = SharedObject.getLocal("messageLSO");
val = (message_l_so.data.val == undefined) ? 0 : message_l_so.data.val;
message_l_so.data.val = val;
message_txt.text = "movie A value: " + val;
```

2. Create a new directory somewhere on your computer. Name this directory *LSOTest*.
3. Create two subdirectories within *LSOTest*. Name these subdirectories *movieAPath* and *movieBPath*.
4. Save the document as *movieA.fl*a to the *LSOTest/movieAPath* directory.
5. Export the movie as *movieA.swf* to the *LSOTest/movieAPath* directory.

6. Create a new Flash document, and on the first frame of the main timeline add the following code (it's almost identical to the code in *movieA.fla*):

```
this.createTextField("message_txt", 1, 100, 100, 100, 20);
message_l_so = SharedObject.getLocal("messageLSO");
val = (message_l_so.data.val == undefined) ? 0 : message_l_so.data.val;
message_l_so.data.val = val;
message_txt.text = "movie B value: " + val;
```

7. Save the document as *movieB.fla* to the *LSOTest/movieBPath* directory.
8. Export the movie as *movieB.swf* to the *LSOTest/movieBPath* directory.
9. Open *movieA.swf* in a web browser and reload the page several times. Each time you reload, you should see the number increment by one.
10. Test *movieB.swf* in the same way you tested *movieA.swf*. You should see the number increment by one with each reload as well. However, notice that the number starts at 0. This is because, as it stands, *movieA.swf* and *movieB.swf* use different shared objects. Even though the shared objects have the same name, they have different paths.
11. To cause both movies to use the same shared object, we must tell them to look in the same path. Modify the line of code that opens the shared object so that it reads as follows in both *movieA.fla* and *movieB.fla*:

```
message_l_so = SharedObject.getLocal("messageLSO", "/");
```

This causes both movies to look to a common path for the shared object, and hence use the same file.
12. Save the Flash documents and re-export the *.swf* files.
13. Test the movies again as in steps 9 and 10. This time, notice that each update to one movie also increments the value used by the other movie. This is because they are now using the same shared object.

See Also

Chapter 17

16.6 Storing Persistent Data on the Server

Problem

You want to store persistent data on the FlashCom server.

Solution

Use a remote shared object to store persistent data on the server and access it from one or more clients. Use the *SharedObject.get()* method in the Server-Side ActionScript to create the shared object, and then connect the client(s) to the shared object

using the *SharedObject.getRemote()* and *connect()* methods in the client-side ActionScript. Or, you can create the RSO entirely from the client using the *SharedObject.getRemote()* method.

Discussion

To create persistent remote shared objects, you can use one of two approaches. Either you can create the server-side shared object using the *SharedObject.get()* method in an *.asc* file and then connect to it from the Flash clients, or you can handle everything from the client. Creating the shared object on the server side is preferable in most scenarios because it allows you to do more things with the data on the server. (The server-side *.asc* file must exist within the FlashCom application. See Recipes 14.1 and 14.14.) The *get()* method requires you to specify the name of the shared object as well as a Boolean value indicating whether the shared object should be persistent (*true*) or not (*false*). (In this case, you want to create a persistent object. See Recipe 17.5 for information about creating nonpersistent shared objects.) If a shared object with the same name already exists on the server, then it is opened; otherwise, a new shared object is created.

```
// Create a new persistent shared object named myFirstRSO and
// assign that reference to a variable named myServer_r_so.
// This code belongs in an .asc file on the FlashCom server.
myServer_r_so = SharedObject.get("myFirstRSO", true);
```

Once a shared object has been created on the server, you can connect Flash movie clients to it using the client-side *SharedObject.getRemote()* and *SharedObject.connect()* methods. The *getRemote()* method requires three pieces of information: the name of the shared object (which must match the name of the shared object that was created using the Server-Side ActionScript), the URI for the net connection to the FlashCom server (which can be retrieved using the *uri* property of the net connection object), and a Boolean indicating whether the shared object is persistent (*true*) or not (*false*). The *getRemote()* method returns a *SharedObject* instance. It is used instead of the *new* operator to instantiate a new remote shared object.

Additionally, you must also instruct the client-side shared object to connect to the FlashCom server to synchronize with the server-side object by using the *connect()* method. The *connect()* method requires that you specify which connection object to use to make the connection.

```
// Create a new, client-side, persistent RSO in the Flash movie. This code looks for
// a server-side shared object named myFirstRSO on the server and application
// specified by the URI (returned by the myConnection.uri property).
myClient_r_so = SharedObject.getRemote("myFirstRSO", myConnection.uri, true);
myClient_r_so.connect(myConnection);
```

When any changes are made to a remote shared object, the server automatically synchronizes the client-side data and invokes the *onSync()* method of the client-side

shared object for all clients. Therefore, to catch any synchronization events, you should define an *onSync()* method for the client-side shared object. For example:

```
myClient_r_so.onSync = function () {
    trace("RSO data has been updated/retrieved.");
};
```

It is important to understand that there will always be some degree of latency between the creation and connection of the remote shared object and the retrieval of the data from the server-side object. Therefore, you cannot rely on the data being available within the client immediately following the calls to *getRemote()* and *connect()*. Consider, for example, that you want to use data from a remote shared object to initialize a list box in a client movie. It might seem natural enough to try something like this:

```
myClient_r_so = SharedObject.getRemote("myFirstRSO", myConnection.uri, true);
myClient_r_so.connect(myConnection);

// Try to set the data provider of the list box to an array stored in the shared
// object (it doesn't work).
myListBox.setDataProvider(myClient_r_so.data.anArray);
```

However, this will not work! The problem is that, most likely, the remote data has not yet been loaded into the client. This is why it is important that you use the *onSync()* method to catch any updates to the client-side data (which includes the initial retrieval of the server-side data). Here is the correct way to implement the preceding example:

```
myClient_r_so = SharedObject.getRemote("myFirstRSO", myConnection.uri, true);
myClient_r_so.connect(myConnection);

// Place the setDataProvider() call inside the onSync() method to ensure that the
// server-side data has been retrieved before trying to use it.
myClient_r_so.onSync = function () {
    _root.myListBox.setDataProvider(this.data.anArray);
};
```

See Also

See Recipe 16.10 for important information on how to store data in the remote shared object. Also refer to Recipe 17.5.

16.7 Saving Remote Shared Object Data

Problem

You want to save the data stored in a remote shared object.

Solution

Define an *onAppStop()* callback handler or use the *SharedObject.flush()* method on the server.

Discussion

When you are working with RSOs, there are really two shared objects: the object as it exists on the server and the object as it exists on the client. And in a scenario in which the shared objects are persistent, there is both a persistent, disk-based version and a working version stored in memory. As the client makes changes to the data in the client-side, memory-based shared object, it is automatically sent to the server, and the data in the server-side, memory-based shared object is synchronized with the client-side data. And once the server data is updated, the *onSync()* method is invoked for all clients connected to the RSO.

```
// Create or open an RSO and connect to it.
my_r_so = SharedObject.getRemote("myFirstRSO", myConnection.uri);
my_r_so.connect(myConnection);

// Add data to the RSO. The data is automatically sent to the server.
my_r_so.data.myData = "Your Value Here.";
```

However, even though the memory-based shared objects are kept in synch with one another, the persistent, disk-based shared objects are not always kept in synch with the current data. This is because file access is a relatively “expensive” task. It requires more time to read and write to disk than to read and write to memory. As such, the default behavior for shared objects is that the data is not written to disk until the memory-based object is about to be deleted. This occurs on the client when the Flash movie is being closed, and it occurs on the server when the application is being shut down. You can read more about how to save the client-side data to disk in Recipe 16.4. For more information on storing server-side data to disk, keep reading.

The RSO data should need to be written to disk only if and when the FlashCom application stops. To tell FlashCom server to write the RSO data to disk when the application stops, define an *application.onAppStop()* method that returns true in your *main.asc* file on the server, as follows:

```
application.onAppStop = function () {
    return true;
};
```

You can also invoke the *flush()* method on the server-side shared object (from the *.asc* file) at any point to explicitly write the data to disk:

```
// Place this in your main.asc file or other included .asc file.
ssSo.flush();
```

See Also

Recipe 16.6

16.8 Checking for Updates to Remote Shared Objects

Problem

You want to be able to efficiently check to see what data has changed when a remote shared object is synchronized.

Solution

Use the *list* parameter that is automatically passed to the *onSync()* method. Loop through the parameter to locate all the elements in which the *code* property has the value indicating the type of update you are interested in.

Discussion

When the *onSync()* method is invoked (see Recipe 16.6), it is automatically passed a parameter that contains information about how the data has changed. The parameter is an array of objects, one for each property of the remote shared object. For example, if a remote shared object has two properties named *myDate* and *myArray* (meaning *my_r_so.data.myDate* and *my_r_so.data.myArray*), then the *list* parameter passed to the *onSync()* method will have two elements. In turn, each element contains three properties: *code*, *name*, and *oldVal*. One element will have its *name* property set to “*myDate*”, and the other will have its *name* property set to “*myArray*”. The *code* property can have the following values for persistent shared objects:

“*success*”

The server-side value was successfully updated from a change made by this client.

“*reject*”

The client tried to make a change to the value, but it was rejected by the server—usually because the same object was being updated by another client.

“*change*”

The client-side value has been updated based on a new value from the server.

“*delete*”

The property has been deleted.

“*clear*”

Either the object is not persistent on the server or on the client, or all the properties have been deleted.

This change information is useful when you want to make changes in the client movie following a synchronization event, and you need to determine what data changed and how. For example, in a whiteboard application, you might use an RSO to store the positions of various movie clips that can be controlled by multiple users. Each movie clip's data can be stored as a different element in the RSO. When one client updates the position of a movie clip, you want to update that movie clip's position in all other connected clients. However, you don't want to unnecessarily update the positions of other movie clips that have not been moved. The following client-side code defines an *onSync()* method that determines which movie clips moved and updates their positions accordingly:

```
myClient_r_so = SharedObject.getRemote("myFirstRSO", myConnection.uri, true);
myClient_r_so.connect(myConnection);

myClient_r_so.onSync = function (list) {
  for (var i = 0; i < list.length; i++) {
    // Synchronize changes to a movie clip's position across all connected clients.
    // Check only for elements marked as "change" because the movie clip's properties
    // will be changed locally already for those marked "success".
    if (list[i].code == "change") {
      // This code assumes that the name of the elements in the shared object
      // correspond to the names of movie clips in _root.
      _root[list[i].name]._x = this.data[list[i].name]._x;
      _root[list[i].name]._y = this.data[list[i].name]._y;
    }
  }
};
```

Of course, the preceding code does not work in isolation. It requires that the client also contain code that writes to the RSO each time a client moves a movie clip. For example:

```
myMovieClip_mc.onPress = function () {
  this.startDrag();
};

myMovieClip_mc.onRelease = function () {
  this.stopDrag();

  // Create a new property in the RSO with the same name as the movie clip. Set that
  // property's value equal to an object with _x and _y properties.
  _root.myClient_r_so.data["myMovieClip_mc"] = {_x: this._x, _y: this._y};
};
```

See Also

See Recipe 16.10 for important information on how to store data in the remote shared object. Also refer to Recipe 17.6.

16.9 Reading Values from a Server-Side Shared Object

Problem

You want to read values from a server-side shared object.

Solution

Use the *SharedObject.getProperty()* method.

Discussion

You cannot read from a server-side shared object the same way you can from a client-side shared object. Use the *SharedObject.getProperty()* method to read from a server-side shared object. This method requires that you specify the name of the property to read, and it returns the value of that property.

```
// This code should be within an .asc file. It retrieves the value of myProperty and
// assigns that value to the variable myPropertyVar.
myPropertyVar = my_r_so.getProperty("myProperty");
```

See Also

Recipes 16.3, 16.8, and 16.10

16.10 Adding Data to a Server-Side Shared Object

Problem

You want to add values to a server-side shared object.

Solution

Use the *SharedObject.setProperty()* method.

Discussion

If you try to add values to a server-side shared object in the same way that you add values to a client-side shared object, you will receive an error. The correct way to add values (or modify existing values) to a server-side shared object is to use the *SharedObject.setProperty()* method. The *setProperty()* method requires that you specify both the name of the property and the value to assign to the property.

```
// This example code should appear in an .asc file. It adds a property named
// myProperty to a shared object and assigns the property the value of 6.
my_r_so.setProperty("myProperty", 6);
```

If you want to modify an existing property relative to its current value, you can use the `getProperty()` method in conjunction with `setProperty()`:

```
// Set the myProperty property to one more than its current value.
my_r_so.setProperty("myProperty", my_r_so.getProperty("myProperty") + 1);
```

See Also

See Recipe 16.9 for details on `SharedObject.getProperty()`. See Recipe 16.2 for important differences when setting data for a local shared object.