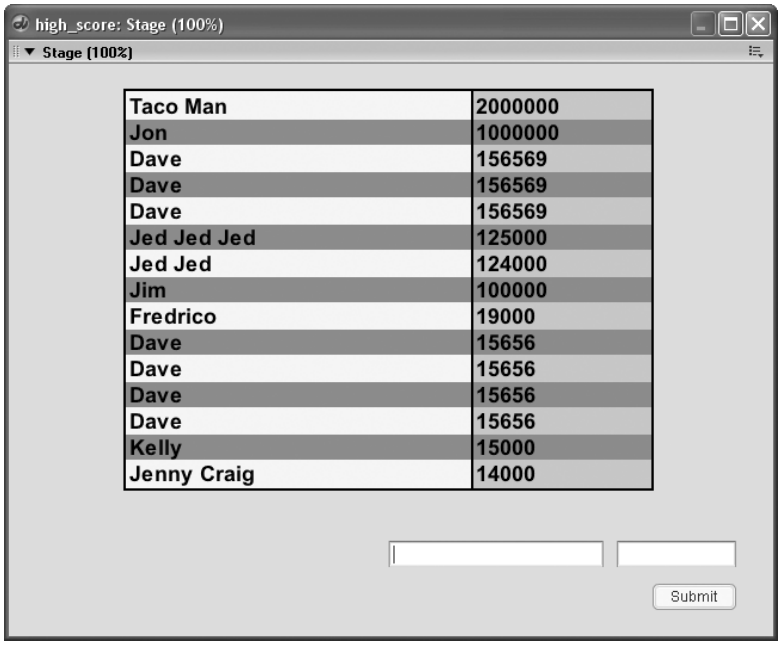
Creating a High-Score Table

A high-score table is an important element in just about any game where the player is rewarded with points. Not only does it give the player something to strive for, it increases the replay value of the game by making the player want to beat their own high score.



The completed High-Score Table

In this lesson you will learn how to create a high-score table for an online game. You'll use Director with PHP and mySQL to allow storing scores, names, levels and anything else you may need. In addition, you'll learn some basic security measures that will keep your game safe from would-be hackers and cheaters.

What You Will Learn

In this lesson you will

- Create a mySQL database on your Web server
- Write PHP scripts to send and get information from mySQL
- Learn more about Object Oriented Programming (OOP)
- Use OOP to create network objects in Lingo
- Use SQL to sort your high-score data
- Create a demo movie to tie it all together
- Learn basic security measures to keep your data private

Approximate Time

This lesson should take you 2 to 3 hours to complete.

Lesson Files

Media Files:

None

Starting Files:

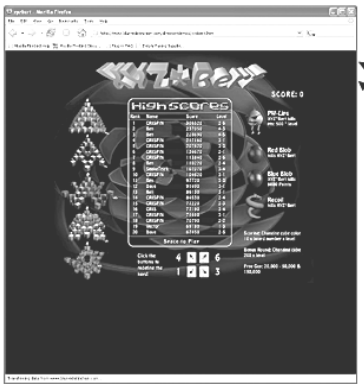
None

Completed Files:

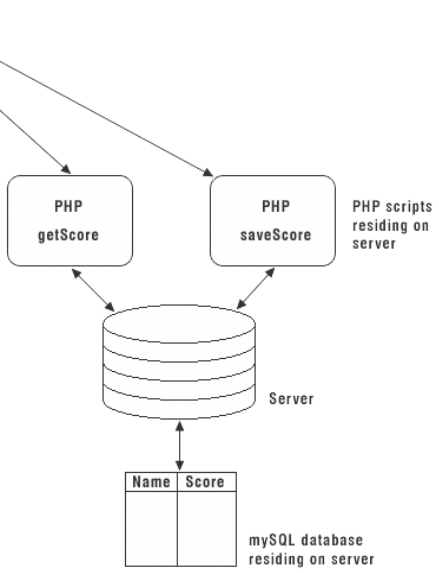
high_score_demo.dir

Getting Started

First, let's talk a little about the overall methodology you'll be using. You are going to have a Shockwave movie running on the user's local machine that communicates with a couple of PHP scripts sitting on your server. Those scripts will allow your Shockwave movie to communicate with a mySQL database, also on the server. The following image should help to illustrate the process.



Shockwave movie running on your local machine



I chose to use PHP and mySQL for two reasons: they are available through nearly every Web hosting company, and they are perfect tools for this lesson.

The first thing you will need to complete this lesson is access to an admin account on a Web server. This is necessary so you can create and administer the mySQL database. You'll also need a program like Macromedia's Dreamweaver in order to upload the PHP scripts and Shockwave movie for testing.

In addition, you will need a tool for administering and working with the database. All host providers that provide mySQL should have such a tool. For this lesson, I will use the popular phpMyAdmin tool, which gives you a nice Web-based interface to administer your databases. Not only is phpMyAdmin a robust and well-developed tool, it used by many host providers, including my own. You can find more information about phpMyAdmin by going to their Web site at: www.phpmyadmin.net

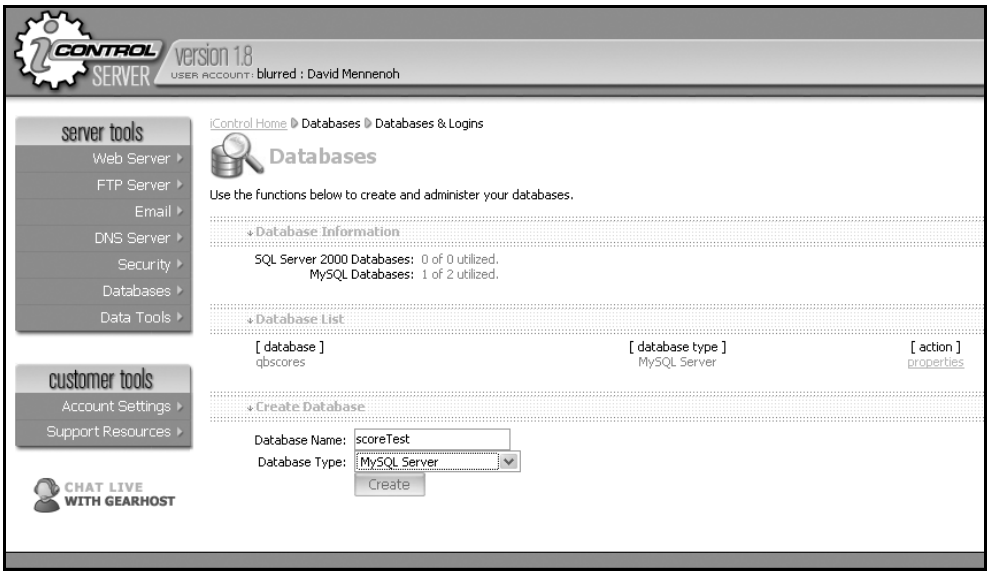
Creating the Database

Before doing anything else, you'll need to log into your hosting account and create a mySQL database and table that will house the high-score data. Recall from Lesson 7, "Implementing a Database," that a single database file can contain a virtually unlimited number of individual tables. This may come in handy as many host providers will limit the number of databases you can use—mine limits me to two.

If you're unable to create an entirely new database, you can create a table in an existing one.

1. Log into your host account and navigate to the Databases area.

Shown here is the Databases page from my account on Gearhost.

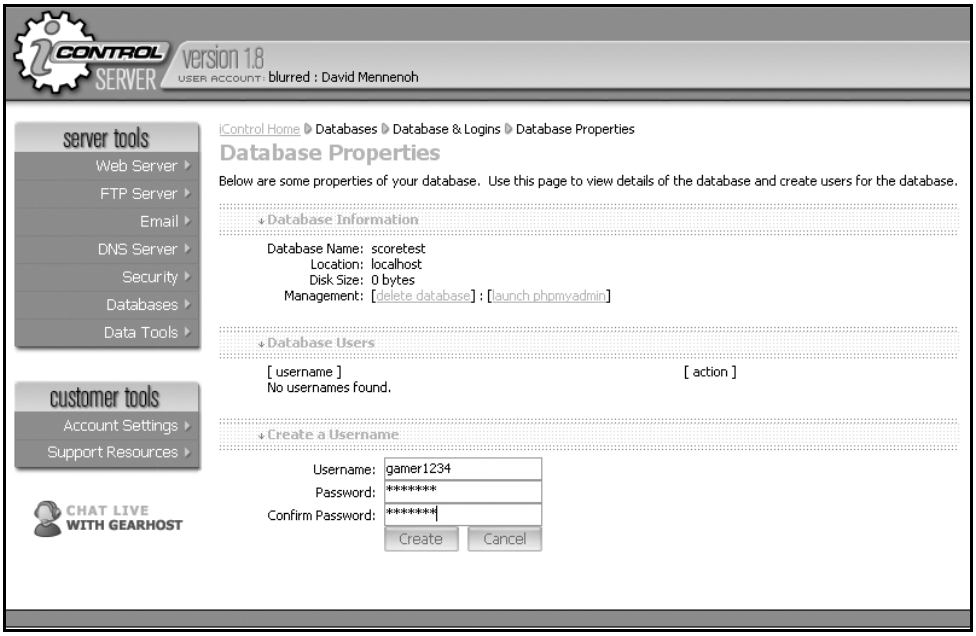


2. Create a mySQL database named *scoretest*.

After you create the database, it will appear in the list of databases for your account.

The next step will be to create a user for your database and launch phpMyAdmin in order to add a table and fields.

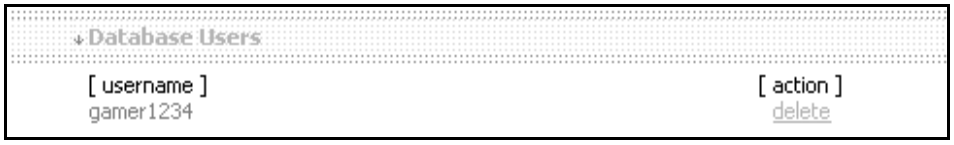
Using my Gearhost account, I click the Properties button next to the name of the database to open the Database Properties page shown here.



On the Database Properties page, you should see some basic information listed about your database. You should also be able to do things like delete the database, add and delete users, and launch phpMyAdmin to do more advanced functions.

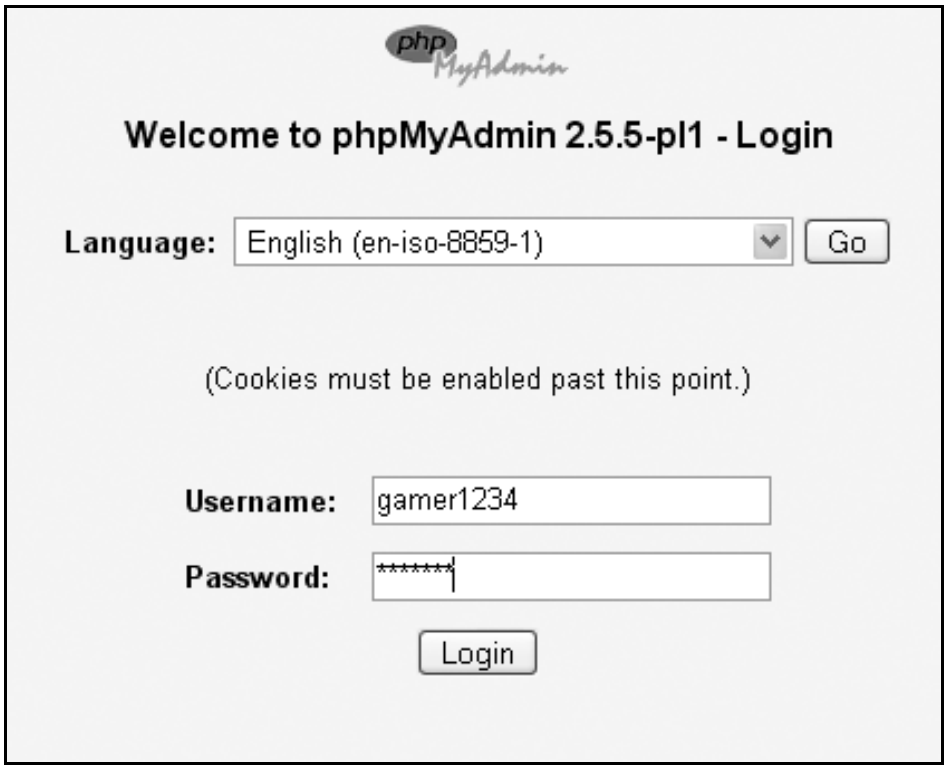
3. Create a user for your database. You can give the user any unique name you like. Here I used *gamer1234* as my user name.

Once your user is created, it will appear in the list of Database Users shown on the page.



You will use this user name and password in your PHP scripts when making calls to the database, so be sure and keep a record of them somewhere. You'll also need the user name and password to get into phpMyAdmin.

4. Launch phpMyAdmin from your Database Properties page, then enter your user name and password on the login page, as shown here:

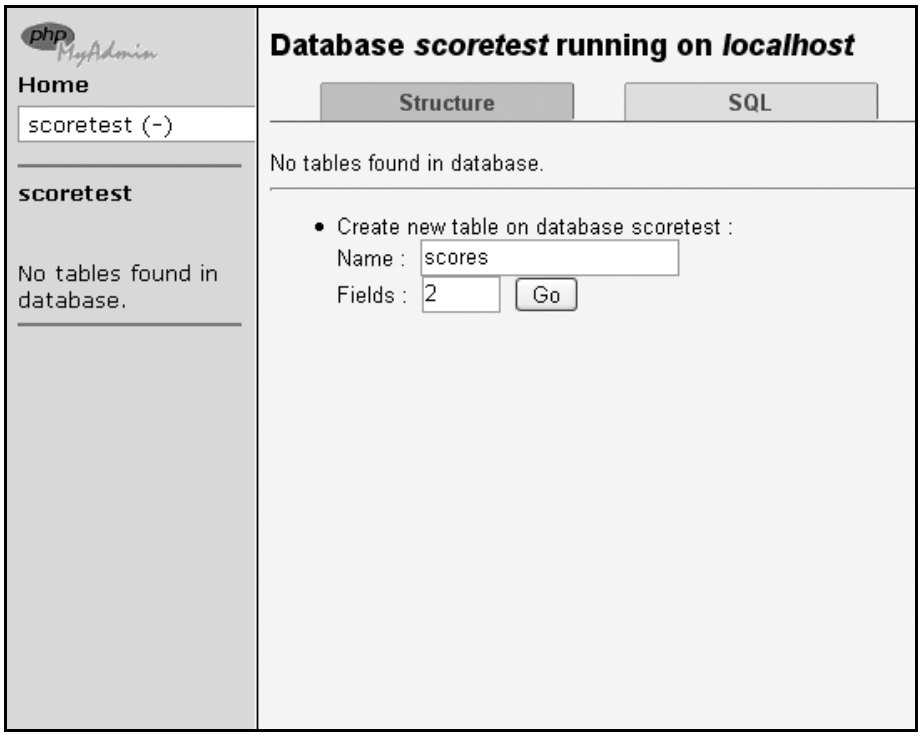


Once phpMyAdmin launches, you'll be prompted to create a table in the empty database.

Creating a Table

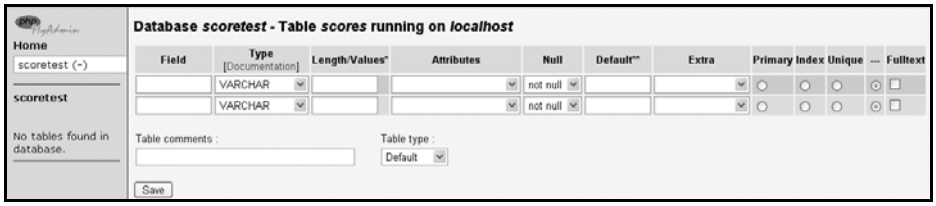
Once the database has been created, you need to create a table in the database. The table, as you should know, is where all of the data is actually stored.

1. Create a table named *scores* with two fields in it.



Fields, or columns, hold the individual pieces of data that form a complete record. In a real game situation you might choose to have fields such as player name, score, level, and board position. For our testing purposes, we'll use just two fields: name and score.

After you click the Go button to create the new table, you should be presented with a screen like the following, where you will define the field names as well as the type of data that will be stored in each field.



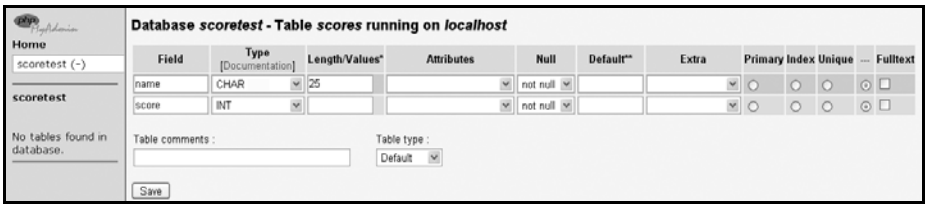
The two fields you'll be using, name and score, will hold string and integer number data, respectively.

2. Enter *name* for the name of the first field and set its Type to Char. Set its length to 25.

The name field will then be able to store up to 25 characters of string data, which should be plenty for most people's names. Compare that with the three characters you used to get in the high-score tables of most arcade video games.

3. Enter *score* for the name of the second field and set its Type to Int.

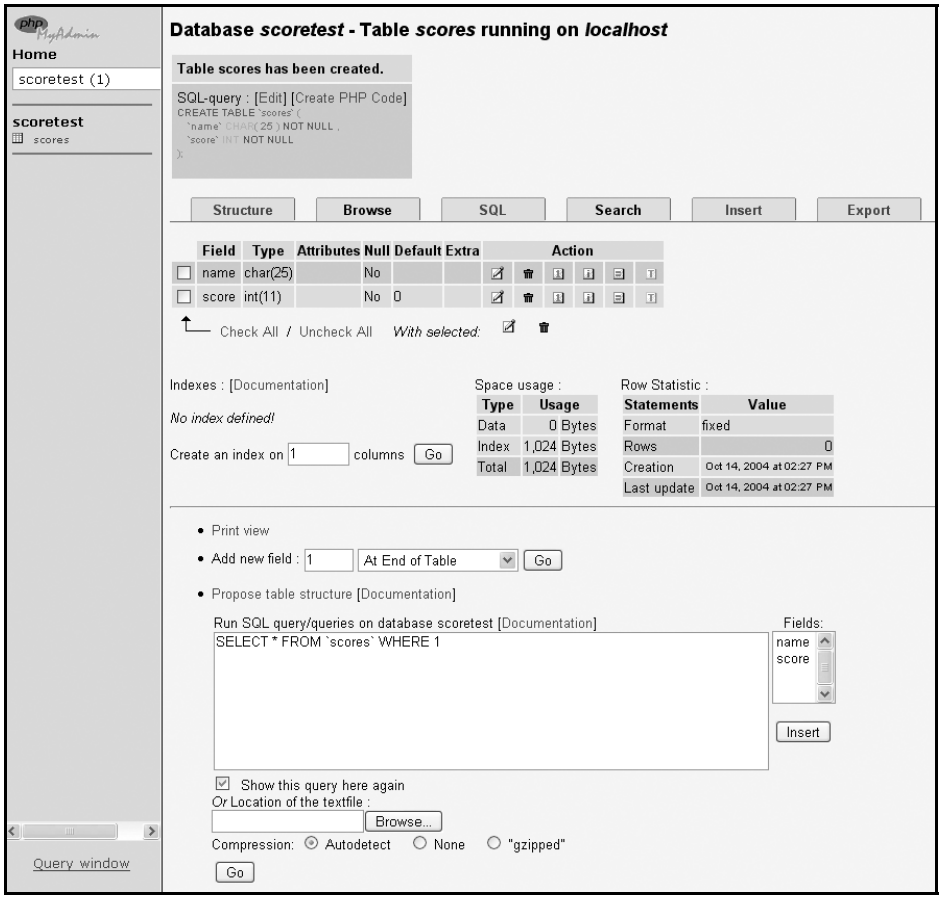
The Int data type will allow you to store a 4-byte integer number that happens to have the same maximum value as Lingo's the maxInteger, which is 2147483647.



Tip: *For more information on the field types you can use, see the mySQL documentation available at* http://dev.mysql.com/doc/mysql*. Section 12 - Column Types will give you detailed information about all the various column types available.*

4. Press the Save button to have phpMyAdmin store the new scores table in the database.

After the table is created and stored in the database, you'll be presented with a screen like this:



From here, you can perform various operations on the table, including adding new fields, running SQL queries, creating indexes, and more.

For our purposes however, the database and table are now ready for use.

5. You can log out of phpMyAdmin by clicking the Home button on the left side, and then clicking the Logout button.

The empty database and scores table are now accessible on your Web server, and can be queried using standard SQL syntax. Of course you must know the user name and password in order to access the data.

Tip: *You should always use some combination of letters and numbers when creating your user name and password. This will at least prevent someone from hacking your data because they know the name of your cat.*

Introducing PHP

PHP stands for Hypertext Preprocessor, although originally it meant Personal Home Page tools. Either way, today PHP is the most widely used server-side scripting language available, outpacing others like ASP, Perl, and Python. This, and the fact that most Web hosting companies provide support for it, led me to choose PHP for this lesson.

Another reason is that unlike other scripting languages, such as Perl, PHP was designed from the ground up to be used on the Web. It has built-in facilities for doing things like sending email, sending or retrieving a document via FTP, and accessing databases. In fact, PHP contains built-in facilities to access mySQL, DBASE, Oracle, InterBase, and others.

For more information about PHP, including complete documentation, you can visit the official Web site at: http://www.php.net/

Creating the Save Script

The save script will accept two incoming pieces of data from Director and then use the SQL command INSERT INTO to insert the data into the scores table.

To create the script you can use any text editor, although using Dreamweaver will be easier since you will need to upload the script to the server once you're finished writing it. Additionally, Dreamweaver will perform syntax coloring on the PHP code, making it easier to read.

1. Open Dreamweaver or your HTML editor of choice and choose an appropriate Web site in which to place your files. Create a new folder in the site's main Web folder, named *testing*.

For ease of use, you should create a testing folder in your main Web folder to keep the scripts and files for the lesson separate from the rest of your site.

2. Create a new document, then choose View > Code.

By default Dreamweaver will place some default HTML code in the new document. Because you'll be creating a PHP-only script, you'll want to erase the default HTML.

3. Select all of the default HTML code and delete it.

Although PHP can be embedded into an HTML page like other scripting languages, you want to create a PHP-only page that will be saved with the extension .php and not .htm.

4. Enter the following PHP code.

```
<?PHP
  $name = $_POST['name'];
  $score = $_POST['score'];

  $link = @mysql_connect("localhost", "gamer1234", "test123");
  if (mysql_errno()){
    exit("-2");
  }
  mysql_select_db('scoretest', $link);
  if (mysql_errno()){
    exit("-3");
  }
  $sql = "INSERT INTO scores(name, score) VALUES('$name', $score)";
  $result = mysql_query($sql, $link);
  if (mysql_errno()){
    exit("-4");
  }
  exit("1");
?>
```

The first thing you should notice is that PHP scripts all begin with the <?PHP tag and end with a corresponding ?> tag, to close the PHP script. The code between the tags is what's executed by the PHP module on the server.

In the first two lines, PHP's automatic global (aka superglobal) $_POST is used to retrieve the data sent in externally—from Director, in our case. What's nice about using $_POST is that it produces an associative array that is nearly identical to a Lingo Property list. Note how you even access the data by name:

```
$name = $_POST['name'];
$score = $_POST['score'];
```

The values contained in the 'name' and 'score' properties in the $_POST array are stored in the PHP variables $name and $score. Note that variables in PHP always begin with a $ identifier. Another similarity to Lingo is the fact that variables in PHP are typeless. You can store any kind of data in a variable at any time.

Don't worry about how the name and score are getting to the PHP script just yet; we'll cover the netLingo necessary for that in a moment.

The next line makes use of mysql_connect to establish a connection to a mySQL server.

```
$link = @mysql_connect("localhost", "gamer1234", "test123");
```

The three parameters being sent to the mysql_connect function specify the server and, of course, the user name and password for the database. Using localhost as the server means the SQL server is running on the machine executing the script—your Web server. Once connected, the link identifier is stored in the $link variable. If any error occurs while attempting to connect to the database server, PHP will return an error string—that will also be returned to Director. By preceding the function name with the @ character, you suppress any error message that might be generated.

The if statement that follows queries PHP's mysql_errno() function which returns False (0) if no error occurred, or an error number if one did occur. If an error occurred connecting to the database, a "-2" will be returned and the script will stop execution. Without the @ sign preceding

the function name, the error string, along with the -2, would also be returned. With the @ sign, only the -2 is returned. Although you could choose to send the error number back, I chose a simpler method, sending simple negative numbers that could be handled in Director.

If the connection is made, the scoretest database is selected using the mysql_select_db function:

```
mysql_select_db('scoretest', $link);
```

This opens the database on the server and makes it the active database, ready for SQL queries to be performed on it. Again, mysql_errno() is checked and if any error did occur a "-3" is returned and the script is halted.

Assuming a successful connection to the server was made, and that the scoretest database is selected, an SQL statement is created and then used in the mysql_query function.

```
$sql = "INSERT INTO scores(name, score) VALUES('$name', $score)";
$result = mysql_query($sql, $link);
```

Here, a standard INSERT INTO command is created that places the values stored in the $name and $score variables into the same named fields in the scores table.

The constructed SQL statement is stored in the $sql variable for use in the query that follows. You may notice the cool way in which PHP can handle variable replacement when constructing the SQL string. Note that no concatenating is necessary; the variables $name and $score appear directly in the string, yet are replaced with their values when interpreted by the PHP parser. This is due to using the double quote. In PHP if you put a variable into a string and the string is wrapped by double quotes, variable replacement will take place and the values will be inserted. If you were to use single quotes instead, no replacement would occur and the literal name of the variable would be used.

The following PHP code shows a simple example of this:

```
$name = "John";
$score = 4420;
$test_one = "$name got $score points.";
$test_two = '$name got $score points.';
echo "$test_one";
echo "$test_two";
```

Output:

```
John got 4420 points.
$name got $score points.
```

Once the SQL statement has been properly constructed and stored in the $sql variable, it is executed on the currently active database by giving it to the mysql_query method. The result of the operation is returned into the $result variable, which will be True (1) on success or False (0) on failure. A final mysql_errno() query is done to see if the Insert operation was successful. If an error occurred a "-4" is returned, otherwise "1" is returned.

That's all there is to the save script for now. In order to test it you'll need to upload it to your Web server. From there you'll be able to call the script from Director.

5. Save the script as scoresave.php - save it into the local testing folder you created earlier. Upload the script to your Web server.

If you're using Dreamweaver, a testing folder on the remote end will be created automatically before the files are uploaded. If you're not using Dreamweaver, be sure the script is uploaded to a remote folder named testing, in your main Web folder.

Note: *As a general rule, most host providers will name your main Web folder www.*

When you're finished, the URL to your script will be:

```
http://www.mydomain.com/testing/scoresave.php
```

With the save script uploaded to the server, you can test it in Director to make sure it's working.

Testing the Script

To test the script you'll need to send it a name and a score that can be inserted into the database. You can do this easily using Lingo's postNetText method right from the Message window.

1. Launch Director and start a new project. Open the Message window.

Using just a few lines of Lingo in the Message window you can send a name and score to the PHP script, and also verify that the script worked.

2. Enter the following Lingo in the Message window:

```
myURL = "http://www.mydomain.com/testing/scoresave.php"
netID = postNetText(myURL, ["name":"Fred", "score":2500])
```

Although you won't see an immediate result, the postNetText command sent the name and score to the specified URL, your PHP script. Notice how a Lingo Property list is used to send the variables and their values. The properties themselves must correspond to the names used in the POST function in the PHP script:

```
$name = $_POST['name'];
$score = $_POST['score'];
```

So now what? The data has been sent to the script, but how are you supposed to know if it actually worked? Notice that you're setting the return of the postNetText command to a variable, netID. When a netLingo method like postNetText is used, a unique network identifier is returned that can be further polled to see if the network transfer is finished and what state it's currently in. Because a network operation runs asynchronously, you will have to wait for it to finish.

3. Enter the following in the Message window:

```
trace(netDone(netID))
```

If all went well you should see a 1 (True), indicating that the network operation completed. However, this only tells you the operation completed—it doesn't tell you if it completed successfully. For that you use the netError() method.

4. Enter the following in the Message window:

```
trace(netError(netID))
```

This time you should see OK returned. If not, you will see one of several possible error codes, such as 4159, which means an invalid URL was specified. You can find a full list of the possible error codes in Director's Help. Look in the Director Scripting Reference > Scripting Objects > NetLingo section, and then click the netError() entry.

Once you know the operation has completed, and that no error occurred you can get the actual result.

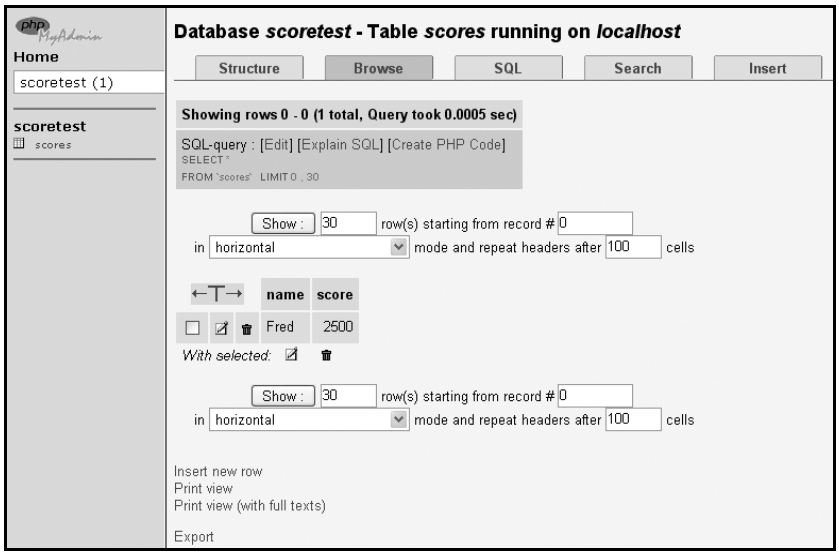
5. Enter the following in the Message window:

```
trace(netTextResult(netID))
```

You should see a "1" returned in the Message window, meaning that the data was successfully added to the database. Recall from the PHP script that a "-2" will be returned if a connection

could not be made, a "-3" if the database could not be selected, and a "-4" if the INSERT command failed.

6. Log into your host admin account and launch phpMyAdmin for the scoretest database. Select the scores table to view the newly inserted data.



As you can see, the name Fred and the score 2500 have been added to the table.

7. Log out of phpMyAdmin.

The next thing we'll do is create the PHP script that will retrieve the scores from the database so that you don't have to go into phpMyAdmin to view the data in the table.

Creating the Get Script

Much like the save script, the get script will need to first log into the SQL server, and then make a connection to the scoretest database, before performing a query. The only real difference in this script is that you will be returning the names and scores from the table, which will require some minor formatting in PHP.

1. In Dreamweaver—or your HTML editor—create a new document. Make sure you're viewing the default HTML code, then select it all and delete it. Enter the following PHP code:

```
<?PHP
  $link = @mysql_connect("localhost", "gamer1234", "test123");
  if (mysql_errno()){
    exit("-2");
  }
  mysql_select_db('scoretest', $link);
  if (mysql_errno()){
    exit("-3");
  }
  $sql = "SELECT * FROM scores";
  $result = mysql_query($sql, $link);
  if (mysql_errno()){
    exit("-4");
```

(continues on next page)

```
  }
  echo "[";
  $row = mysql_fetch_array($result);
  while ($row){
    echo "[\"".$row["name"]."\",".$row["score"]."]";
    $row = mysql_fetch_array($result);
    if ($row){
      echo ",";
    }
  }
  echo "]";
?>
```

The first sections of the script are identical to the save script, connecting to the SQL server, and then selecting the scoretest database. Once the database has been selected, a standard SQL SELECT command is issued using the mysql_query method of PHP. In the form used here—SELECT * FROM scores—all the data in the scores table is selected and returned in the $result variable.

If no error occured, a while loop executes that pulls the individual rows out of the result and creates a Lingo list to be returned to Director. In the main list, each name and score will be placed into a sub-list, making for easy extraction in Lingo. Let's take a closer look at the while loop to see how it works:

```
echo "[";
$row = mysql_fetch_array($result);
while ($row){
  echo "[\"".$row["name"]."\",".$row["score"]."]";
  $row = mysql_fetch_array($result);
  if ($row){
    echo ",";
  }
}
echo "]";
```

Notice the use of the PHP echo command. The echo command lets you output text—or send text back to Director, in our case. You should be aware that unless you explicitly echo a new line character, all of the echoes will be returned as a single string to Director, which is exactly what we want.

First, a single opening bracket is echoed. This will serve as the beginning of the main list. Next, the PHP command mysql_fetch_array() is used. This takes a single row of data from the result and returns an associative array, which is very much like a Lingo Property list.

The while loop then tests to see if the $row variable has data in it. If it does another echo statement is executed:

```
echo "[\"".$row["name"]."\",".$row["score"]."]";
```

This may look complex, but it's actually quite simple. First, the period is PHP's concatenation symbol. With that in mind, the single echo could be broken down into five separate echo statements as follows:

```
echo "[\"";
echo $row["name"];
echo "\",";
echo $row["score"];
echo "]";
```

Except for the first and third lines, the rest should be fairly self-explanatory. The first and third lines, however, use PHP's method of escaping characters by using a backslash. Escaping the double quote character: \" allows the quote to be part of the returned string instead of terminating the string as it normally would. Can you see how this works?

Assuming that the name Fred with a score of 2500 is in the database, the five lines would produce the following:

String after line 1: ["

String after line 2: ["Fred

String after line 3: ["Fred",

String after line 4: ["Fred",2500

String after line 5: ["Fred",2500]

After the echo statement is executed, another call to mysql_fetch_array is made, and again the result is stored in the $row variable. If $row is not empty, another record exists in the database and a single comma is echoed in order to separate the list items. The while loop then executes again, pulling the data from the $row variable and creating another list from it. Note that this works because successive calls to mysql_fetch_array automatically move the row pointer ahead in the database.

This continues until all the data in the result has been read and the $row variable comes up false. When that happens, the while loop terminates and the final closing bracket is added to the end of the string.

Let's take a look at one more quick example, using a few sets of data, to be sure this all makes sense. First, assume the database contains the following data in the scores table:

Fred	2500
Dave	3000
Billy	6700

After executing the mysql_query command, the $result variable would point to the full set of all three names and scores. Next, the first echo is issued, which places an opening bracket into the returned string:

final string: [

The mysql_fetch_array command then places the first row of data into the $row variable, which is tested at the start of the while loop. Because $row is not false, the echo statement in the loop executes, adding to the final string:

final string: [["Fred",2500]

Next, another mysql_fetch_array is issued, placing the second row of data into the $row variable. Because $row is not false, a comma is echoed, changing the final string to:

final string: [["Fred",2500],

The while loop then repeats, again checking if $row is false. Because it isn't, the echo statement executes again, placing the second row of data into the final string:

final string: [["Fred",2500],["Dave",3000]

Once again, mysql_fetch_array is issued, which places the third row of data into the $row variable. A comma is added to the final string and the while loop starts over, and echo statement is run. After this third pass through, the final string will appear as:

final string: [["Fred",2500],["Dave",3000],["Billy",6700]

Again, mysql_fetch_array is issued, but this time there is no data to be returned. Therefore, the $row variable becomes false and the comma isn't added to the end of the final string. When the while loop starts over $row is false, and it immediately exits. Finally, the closing bracket is echoed, producing the following:

final string: [["Fred",2500],["Dave",3000],["Billy",6700]]

The PHP script then stops executing, and the final string is returned to Director where a simple call to the value function will turn the string into a valid Lingo list.

That's enough explanation of how the script works. Let's test to be sure it's working as expected.

2. Save the script as scoreget.php. Save it into the local testing folder, then upload the script to your Web server.

When complete, the URL to this script will be:

```
http://www.mydomain.com/testing/scoreget.php
```

Testing the Script

To be sure the script is functioning properly, you should test it in Director. Although there's currently just one name and score in the database, that will be fine for this test.

1. In Director, open the Message window and issue the following:

```
myURL = "http://www.mydomain.com/testing/scoreget.php"
netID = getNetText(myURL)
```

Instead of the postNetText method you used previously, Lingo's getNetText method is used when no variables are being passed to the script. Don't think that post is only for sending, and get only for retrieving. In fact either method can be used to both send and retrieve data.

2. Test to see if the network operation has completed by issuing the following:

```
trace(netDone(netID))
```

If the operation completed, you will see a 1 (True) output in the Message window.

3. Check for a network error before retrieving the result:

```
trace(netError(netID))
```

If no error occurred, you will see OK in the output.

4. Use the netTextResult method to retrieve the data returned from the script:

```
myData = netTextResult(netID)
trace(myData)
```

In the output pane of the Message window, you should see the following:

```
-- "[["Fred",2500]]"
```

As you can see, the result is currently a string and not a list. That can be easily remedied using Lingo's value method.

5. Enter the following in the Message window:

```
myData = value(myData)
trace(myData)
```

Now, a proper Lingo linear list is output:

```
-- [["Fred",2500]]
```

The data can now be parsed using regular list techniques:

```
trace(myData[1])
-- ["Fred",2500]
trace(myData[1][1])
-- "Fred"
trace(myData[1][2])
-- 2500
```

With the database setup, and the PHP scripts in place, you can create the Lingo parent scripts that will be used to post and retrieve the high-score data from the database.

Using OOP Techniques

In this section you'll make use of Lingo's parent script type to create asynchronous network transfer objects. Although that may sound a little intimidating, these parent scripts will make working with the database quick and easy.

Let's first discuss the reasoning for using OOP. Mostly it's because you need the practice, but also because using an object to monitor a network operation is a good application of objects.

As you should know, you've been using OOP techniques all along. Behaviors themselves are objects, as they are instances of a single script attached to a sprite or frame. A parent script, while very much like a behavior, differs in how it's instantiated. And because a parent script is instanced only into RAM, i.e. it's not attached to a sprite or frame, the instanced object doesn't receive standard frame events like enterFrame and exitFrame.

However, if the object is placed into Director's actorList, it will receive the special stepFrame event that is unique to objects. As long as the object is in the actorList it will receive stepFrame events. To have it stop receiving the events, you simply remove it from the actorList. You'll make use of the actorList and the stepFrame events to have your scripts automatically monitor the state of the network activity.

Introducing OOP

Before we get to creating the actual network objects, let's look at some simple OOP examples.

1. Open Director if it's not already open, and create a new movie. Select the first empty cast member in the internal cast, and press Ctrl/Command+0 to open a script window. Use the Property inspector's Script tab to set the script type to Parent.

2. Enter the following script:

```
on new me
  return me
end
on output me
  trace("Hello World!")
end
```

First, note the use of the new handler. It is calling this handler that instantiates the script into a *child object*, so named because it is created from a *parent* script. Before Director 4, parent scripts were known as "factories" and child objects were "birthed." Other languages, like JavaScript, use the term "class" instead of "parent" script to refer to the same thing.

In the new handler, there only has to be one line: return me. Although there can be more (you can initialize variables, etc.) the only thing that must be included is the return me. What return me does is pass back a reference to the memory address where the script has been instantiated into. By storing this reference in a variable, you can access the scripts methods by referring to the variable. This is also what allows you to instantiate as many copies of the script as you need. Each one is stored in a unique memory location, so you can refer to each instance by using a unique variable. Let's see how this works.

3. Name the script *test* and close the Script window. Open the Message window and enter the following:

```
a = script("test").new()
b = script("test").new()
```

You've just created two instances of the test script and stored references to each script in the two variables, a and b.

4. Enter the following in the Message window:

```
a.output()
```

Probably as expected, you see "Hello World!" in the Message window's output pane. Of course, you can also call the output method in b, but you'll only see the same "Hello World!" message. Let's fix that.

5. Open the test script from the internal cast and then modify the script so that it appears like this:

```
property myColor

on new me, col
  myColor = col
  return me
end

on output me
  trace("Hello" && myColor && "World!")
end
```

Like behavior scripts, parent scripts can also make use of properties that are unique to each instance of the script. Here, you added the myColor property and assigned it to the col parameter being passed in to the new handler. Let's see how this works.

6. Close the Script window and open the Message window. Enter the following:

```
a = script("test").new("red")
b = script("test").new("blue")
```

As before, you created two instances of the script and stored them in the variables a and b. This time however, you passed in "red" and "blue" to the new handler of each script, which is assigned to the myColor property in the script.

7. Enter the following in the Message window:

```
a.output()
```

You see:

```
-- "Hello red World!
```

Now call the output method in b:

```
b.output()
```

You see:

```
-- "Hello blue World!"
```

Even though the objects have been created from the same parent script, they have unique values for their properties. Each object's properties can also be modified at run time, allowing you to do some very powerful things once you grasp their potential.

If you'd like to learn more about Object Oriented Programming techniques, the following Web sites contain some great information.

Irv Kalb's Lingo Object Oriented Programming Environment (LOOPE) Web site features an e-book containing 15 chapters on using objects in Director.

http://www.furrypants.com/loope/

Brennan Young's Invaders is a tutorial on building a Space Invaders clone using OOP techniques.

http://brennan.young.net/Edu/Lingvad.html

J.M. Harward has a nice article on Object Oriented Fundamentals on his Animation Math in Lingo site.

http://www.jmckell.com/OOfun.html

Now let's create the script that will communicate with the PHP get script in order to retrieve the high-score data from the database.

Creating the Get Object

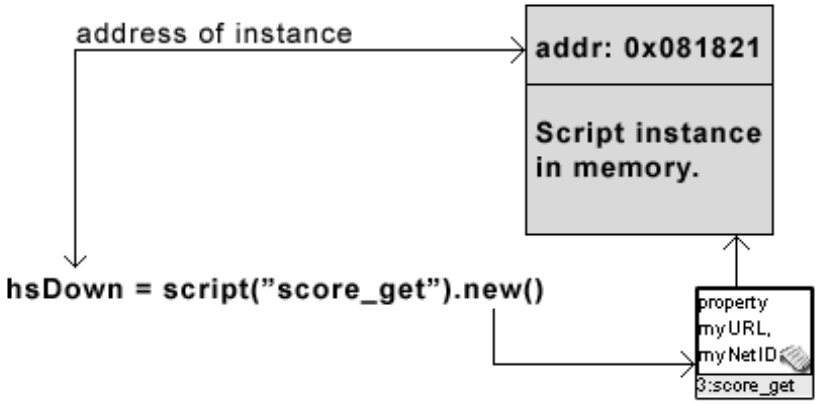
As mentioned earlier, network operations are asynchronous: once you start one, you must wait for it to finish before you can use any returned data from the server. Using a parent script, and the special stepFrame event, will allow your object to essentially monitor itself. By using an object, and being able to create an instance of the object whenever anytime you like, you are freed from the score. You can start a download of your high-score data at any time.

1. Select the next empty cast member and press Ctrl/Command+0 to open a Script window. Use the Property inspector's Script tab to be sure the script type is set to Parent. Enter the following Lingo:

```
property myURL, myNetID, myDone, myErr, scores

on new me
  myURL = ""
  myErr = 0
  myDone = False
  scores = []
  return me
end

on setURL me, scriptURL
  myURL = scriptURL
end

on downloadScores me
  myErr = 0
  myDone = False
  scores = []
  myNetID = getNetText(myURL)
  _movie.actorList.add(me)
end

on stepFrame me
  if netDone(myNetID) then
    myErr = netError(myNetID)
    if myErr = "OK" then
      scores = netTextResult(myNetID)
    end if
    myDone = True
    _movie.actorList.deleteOne(me)
  end if
end

on getHighScoreList me
  return value(scores)
end

on getErr me
  return myErr
end

on isDone me
  return myDone
end
```

2. Name the script *score_get* and close the Script window.

Before doing any testing, let's see exactly how the script works. First, several properties are declared and initialized to default values in the new method. The required return me is then executed, which ends the new method and returns the memory address of the script instance. The following image should help to illustrate this process.



The setURL method allows you to set the URL to the PHP script. Although you could hard-code the URL into the object, by setting it externally you help to generalize the object and make it easier to reuse for other projects. It's up to you to properly set the URL before calling downloadScores, although no error will occur if you don't. The object will fail silently, and you can get the error by calling the object's getErr method.

The downloadScores method resets the properties to default values, checks to see if the URL has been set, and then initiates a network transfer using getNetText. Notice you're not resetting myURL, however. This will allow you to download the score data as many times as needed without having to set the URL each time. Once the transfer is initiated, the object adds itself to the actorList so that it will receive stepFrame events.

Note: *While you can call the methods of an object even when the movie is stopped, the* stepFrame *handler, like other frame events, will only be executed while the movie is playing and the object is in the* actorList*.*

With the object in the actorList, the stepFrame handler will be run at the beginning of each frame event.

Tip: *The* stepFrame *event is actually sent before the frame's* prepareFrame *event. Both occur before the frame has been drawn.*

At each frame update, a call to netDone(myNetID) is performed, and will return True when the network operation has finished. Once the operation is finished, any returned error condition is retrieved and stored in myErr. If an OK is returned, then the high-score data is retrieved by calling netTextResult(myNetID), and stored in scores. Finally, myDone is set to True and the object is removed from the actorList by using the deleteOne method available to lists. With the object removed from the actorList it will still exist as it did before, but it won't be processing stepFrame events and taking up processor slices.

Note: *Having an object remove itself from the* actorList *in the* stepFrame *handler used to cause Director to crash. This limitation no longer exists, and you can now safely have objects remove themselves.*

After the stepFrame handler are three one-line methods that allow you to poll the various properties of the object. You'll use these "accessor" methods from outside the object in order to

know when the network operation is finished, what the error condition is, and also to retrieve the high-score data.

Let's do a little test to see how the object is working.

Testing the Object

To test the object, you'll need a simple loop on frame behavior so the movie can be played.

1. Double-click in the behavior channel at frame 5 and create a loop on frame behavior:

```
on exitFrame me
  _movie.go(_movie.frame)
end
```

2. Name the script *loop_on_frame*, then close the Script window. Play the movie.

With the movie playing, you can create an object from the parent script and then initiate a download of the high-score data from your server.

3. Open the Message window and enter the following:

```
hsDown = script("score_get").new()
hsDown.setURL("http://mydomain.com/testing/scoreget.php")
hsDown.downloadScores()
```

First you created an instance of the score_get script and stored a reference to it in the hsDown variable. Next you called the setURL method so the object will know the URL to the PHP script. The downloadScores method is then called, which initiates the network transfer.

4. Enter the following:

```
trace(hsDown.isDone())
```

You should see a 1 (True) output in the Message window, indicating the transfer has finished.

5. Check for an error:

```
trace(hsDown.getErr())
```

You should see:

```
-- "OK"
```

If you don't get OK returned, you'll get one of several different error code numbers, most of which are listed as follows:

```
4: Bad MOA class. The required network or nonnetwork Xtras are improperly installed or not installed at all.
5: Bad MOA Interface.
6: Bad URL or Bad MOA class.
20: Internal error.
900: File is Read Only
903: Disk is Full
905: Bad Filespec
2005: Incompatible Net Xtras
2018: postNetText used with no variables - use getNetText.
4146: Connection could not be established with the remote host.
4149: Data supplied by the server was in an unexpected format.
4150: Unexpected early closing of connection.
4152: Failed network operation
4154: Operation could not be completed due to timeout.
```

(continues on next page)

4155: Not enough memory available to complete the transaction.
4156: Protocol reply to request indicates an error in the reply.
4157: Transaction failed to be authenticated.
4159: Invalid URL.
4164: Could not create a socket.
4165: Requested object could not be found (URL may be incorrect).
4166: Generic proxy failure.
4167: Transfer was intentionally interrupted by client.
4240: The network xtras weren't initialized properly.
4242: Download stopped by netAbort(URL).
4836: Download stopped for an unknown reason, possibly a network error, or the download was abandoned.

You can choose whether to display these messages or simply show something like "Network Error, please try again." I typically choose the latter.

6. Show the score data:

```
trace(hsDown.getHighScoreList())
```

You should see:

```
-- [["Fred", 2500]]
```

As you can see, a standard linear list is returned that will make it very easy to remove the names and scores from the display.

7. Stop the movie and enter the following in the Message window:

```
trace(hsDown)
```

You'll see something like:

```
-- <offspring "score_get" 2 3b67c48>
```

This shows you that the hsDown variable contains a reference to an instance that the score_get script located at memory address 3b67c48. If you want to dispose of the object so that you save memory when it's not needed, simply set any references to the object to 0. When there are no more references, Director will automatically garbage collect it, and the memory it was using will be freed. In this case, with a single download object, there's not much need to dispose of it. But in a game when you might have many objects created from a single parent script, disposing of objects when they're not in use can save a lot of memory.

8. Enter the following in the Message window:

```
hsDown = 0
```

The object will then be removed from memory because the variable is no longer referencing it. Note that when you add an object to the actorList you also create another reference to the object. In this case if you set the variable reference to 0 the object would still remain "alive" because of the actorList reference to it. That is why it's nice to be able to have the object automatically remove itself from the actorList.

Let's create the save object now.

Creating the Save Object

Based on the testing you've already done, and the fact that the save object is very much like the get object, you probably could write this script without any help. The only real difference is that here you'll pass in a name and score, and use the postNetText method to send the data to the PHP script.

1. Select the next empty cast member and press Ctrl/Command+0 to open a Script window. In the Property inspector's Script tab, be sure the script type is set to Parent. Enter the following Lingo:

```
property myURL, myNetID, myDone, myErr, phpErr

on new me
  myURL = ""
  myErr = 0
  phpErr = 0
  myDone = False
  return me
end

on setURL me, scriptURL
  myURL = scriptURL
end

on postScore me, nam, scor
  myDone = False
  myErr = 0
  myNetID = postNetText(myURL, ["name":nam, "score":scor])
  _movie.actorList.add(me)
end

on stepFrame me
  if netDone(myNetID) then
    myErr = netError(myNetID)
    if myErr = "OK" then
      phpErr = value(netTextResult(myNetID))
    end if
    myDone = True
    _movie.actorList.deleteOne(me)
  end if
end

on getErr me
  return myErr
end

on getPhpErr me
  return phpErr
end

on isDone me
  return myDone
end
```

As you can see, much of the script is just like the score_get script, though there are some notable differences.

First the phpErr property will allow you to check the error code returned from the PHP script. This is in place because even if the object finishes the post operation and returns OK, there might have been a problem on the server end. Recall that a 1 is returned from the PHP script if the SQL

INSERT operation was successful. To verify the data was posted you should not only call getErr to make sure OK is returned, you should then call getPhpError to make sure 1 is returned.

To post a new high score to the database, you call the object's postScore method, sending in the name and score as parameters. For example, if your object is named hsUp you could post a new score for Billy using

```
hsUp.postScore("Billy", 1250)
```

This, of course, starts a network transfer by issuing the postNetText command. The object is then placed into the actorList in order to be able to receive stepFrame events.

As before, once the object is in the actorList, the script's stepFrame handler is executed and runs until netDone(myNetID) returns true. Once the network transfer is complete, the error codes are retrieved and the object removes itself from the actorList to prevent the stepFrame handler from executing unnecessarily.

The final three methods in the script allow you to retrieve the error codes, as well as see if the network transfer is complete.

2. Name the script *score_save* and close the Script window.

Now let's add a few more names and scores to the database in order to test the script.

3. Save the movie as *high_score* before continuing.

If you completed the other lessons in the book, you should have a dmx2004_source folder on your hard drive containing your project folders. You can either save the file into the root of the dmx2004_source folder, or you can create a project_five folder to save the movie into.

Testing the Object

In order to test the save object, let's post a few different names and scores to the database. You can then use the get object to retrieve the scores, making sure everything is working before we move on to displaying the information.

1. Play the movie, then enter the following in the Message window:

```
hsUp = script("score_save").new()
hsUp.setURL("http://www.mydomain.com/testing/scoresave.php")
hsUp.postScore("Billy", 1750)
```

If all went well, the name and score have been added to the database table. You can test to be sure using the getErr and getPhpErr methods of the object.

2. Enter the following in the Message window:

```
trace(hsUp.isDone())
```

When the object is finished with the network operation, you will get a 1 in the Message window's output pane. Once the operation is done, you can check the error state.

3. Enter the following:

```
trace(hsUp.getErr())
```

You should see:

```
-- "OK"
trace(hsUp.getPhpErr())
```

You should see:

```
-- 1
```

If getErr returns OK, and getPhpErr returns 1, you can be confident the score was posted to the database. Let's enter another name.

4. Enter the following in the Message window:

```
hsUp.postScore("Jill", 2750)
```

You should make sure the operation has completed, by calling the isDone method, and also check for any errors. Once the upload has completed, go ahead and upload one or two more names and scores.

Once you've finished posting new names and scores to the database, get the list by creating a new instance of the score_get parent script.

5. In the Message window, download the list of high scores by entering the following Lingo:

```
hsDown = script("score_get").new()
hsDown.setURL("http://www.mydomain.com/testing/scoreget.php")
hsDown.downloadScores()
```

6. Enter the following to see if the download has completed, then output the list of scores:

```
trace(hsDown.isDone())
```

Once True is returned, you can output the score list:

```
trace(hsDown.getHighScoreList())
```

Depending on what names and scores you posted, you'll get something like this:

```
-- [["Dave", 2750], ["Billy", 1750], ["Pete", 1000], ["Fred", 2500]]
```

Notice I didn't have you check the error condition before getting the high-score list. That's because when you initiate the transfer, by calling downloadScores, the scores property in the object is set to an empty list: []. If the download is completed and you get an empty list returned when you call getHighScoreList, you can then call the getErr method and decide what to do. This way, you only check the error condition when an error has occurred.

7. Stop the movie before continuing.

Examine the list that was returned from the database:

```
[["Dave", 2750], ["Billy", 1750], ["Pete", 1000], ["Fred", 2500]]
```

If you'll be displaying these in a high-score table, the order should be dictated by the scores, going from highest to lowest. Currently, they are in the order in which they were added to the database. Although you could write Lingo to sort the list, it wouldn't be the simplest solution. Instead, you can make a simple change to the SQL statement that selects the data and have it sent back in the order you want.

Modifying the SQL SELECT

In order to have the high-score data returned in the proper order, you can tell SQL to select the data in whatever order you specify, according to a particular field. In our case we want the results sorted in descending order, according to the values in the score field of the table. In order to do this you'll need to modify the scoreget.php script, and then re-upload it to your server.

1. Keep Director open and also open Dreamweaver, or your HTML editor.

By keeping Director open, the hsDown object you created will still be available, making it quick and easy to test the results of modifying the PHP script.

2. Open the scoreget.php script from your testing folder. Modify the line of code that creates the SQL string and assigns it to the $sql variable. Modify it so the SELECT statement is as follows:

```
$sql = "SELECT * FROM scores ORDER BY score DESC";
```

All you really did was add the ORDER BY score DESC onto the end of the current SELECT statement. This will cause the data to be sorted on the score field in descending order.

3. Save the script, then upload it to your server. Close your HTML editor and return to Director and press Play. In the Message window enter the following:

```
hsDown.downloadScores()
trace(hsDown.isDone())
```

Once you get True returned from isDone, you can display the high-score data:

```
trace(hsDown.getHighScoreList())
-- [["Dave", 2750], ["Fred", 2500], ["Billy", 1750], ["Pete", 1000]]
```

As you can see, the same data is now listed from high score to low score—perfect for displaying.

4. Stop Director before continuing.

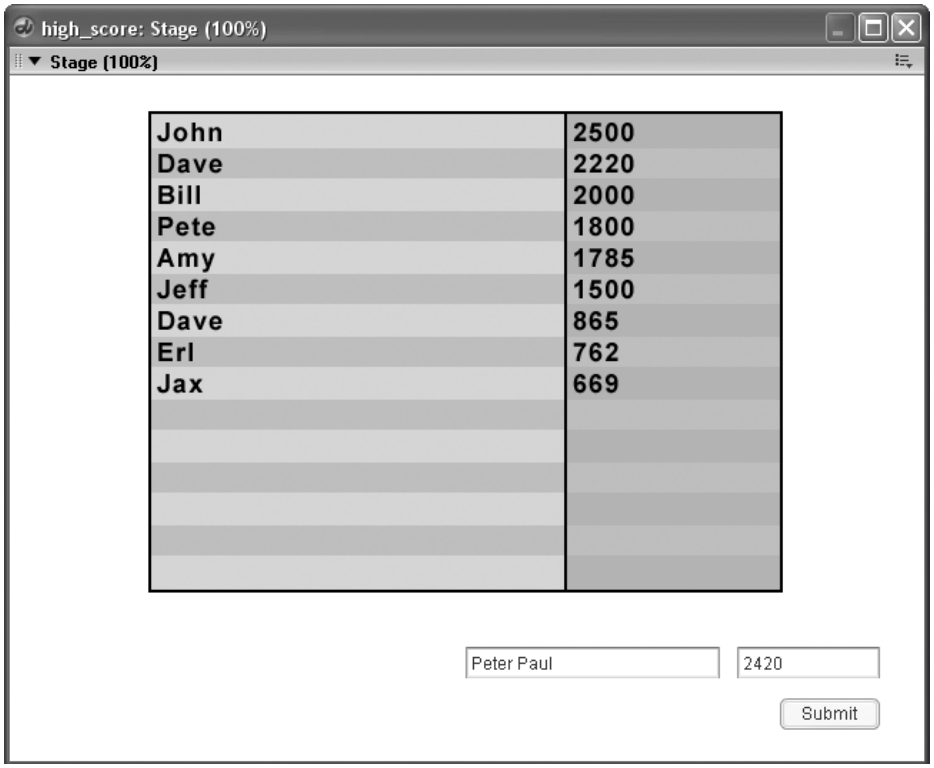
At this point, the server-side mySQL database and the PHP scripts are working properly. You've also created two Lingo objects that will let you communicate with those PHP scripts in order to send and receive high-score data.

It's now time to take the pieces you've assembled so far and create a demo movie with them.

Creating a Demo Movie

In this section, you'll add a text field, buttons, and code to the high_score movie in order to create a working demo. This demo will display the high scores from the database as well as allow you to submit new score data without having to use the Message window.

When the movie is completed you'll have something similar to this:



Creating the Score Table

Currently, you should have a nearly empty Score, consisting of a single loop on frame script attached to frame 5. Let's begin by creating the text member to display the score data.

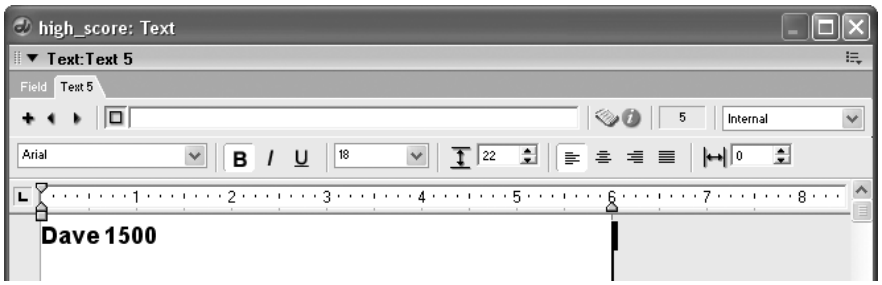
1. Click in the empty Stage and then choose the Movie tab in the Property inspector. Set the Stage size to 640 x 480, and choose a solid white background.

This will ensure that what you see on screen will look similar to the screenshots in this lesson.

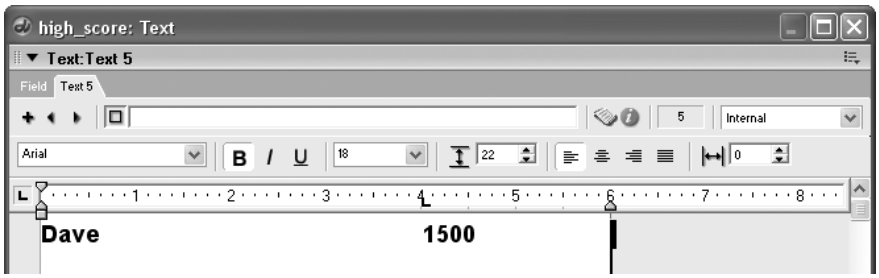
2. Choose the Text tool and drag out a text box nearly as wide as the Stage. Enter *Dave* and then press the Tab key. Enter *1500*.

You can actually enter any name and score you like; these only serve as placeholders so you can get a feel for the placement. Initially, you won't see much separation between the name and the score, caused by the tab, but that will change in a moment.

3. Double-click the new text cast member, in the internal cast, to open the member for editing. You will see something similar to the following:



4. Single click in the ruler at approximately the 4-inch mark, to set a tab:



As soon as the tab is set the score jumps to it, because of the Tab character you entered when typing the data. You'll make use of Lingo's TAB constant when constructing the display data, to make the score column line up nicely no matter how long a particular name might be. There are other ways you could format the data, such as using two separate text members, but this simple approach will do fine for this demo.

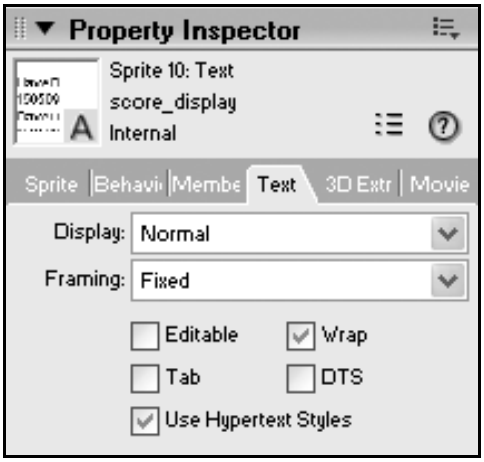
5. Name the member *score_display* and then press Ctrl/Command+A to select all of the text. Change the font and size to something appropriate; I used 18 point Arial, set to Bold.

If you were going to be using this in a real game, you'd likely embed and use a more "fun" font rather than using Arial bold. But again, this is a demo; you can customize it later as you see fit.

6. Close the Text window and adjust the text sprite's span in the Score so it occupies frames 2 through 5 of channel 10.

Placing the sprite in channel 10 will allow you to place background graphics behind the text.

7. Select the text sprite and choose the Sprite tab in the Property inspector. Set the ink type to Background Transparent. Choose the Text tab and set the Framing to Fixed.



By default, the framing is set to Adjust to Fit, which will cause the sprite to expand to whatever size it needs to accommodate the text. By setting it to fixed, it will retain the size you set it to, regardless of how much text you place into it.

8. Return to the Sprite tab in the Property inspector and set the width of the sprite to 432 and the height to 337. Place the Sprite at X: 100 and Y: 25.

Setting the sprite to this size will allow you to display 15 high scores, if you're using 18pt. Arial. If you're using another font, or size, you can enter additional lines of text into the member in

order to set its size properly. You'll be adding a behavior to the sprite that will erase any text on beginSprite, so feel free to add as much setup text as necessary.

Once you've got the text sprite sized properly, you can add background elements to make it more readable and better looking.

9. Use the filled rectangle and line tools to add background elements to the score table. Use sprite channels 1 though 9 for this.

In the following image I used two large filled rectangles to define the name and score areas. Then I placed several thin horizontal rectangles in order to make each entry appear on its own line, making it easy to read. Finally, I placed an unfilled rectangle to create the border, and used the Line tool to make the vertical line between the names and scores.

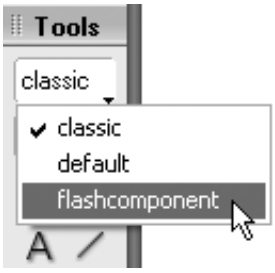
Jon	1000000
Dave	156569
Dave	156569
Dave	156569
Dave	15656
Dave	15656
Dave	15656
Dave	15656
Kelly	15000
Dave	2750
Fred	2500
Billy	1750
Pete	1000

Once you're satisfied with the table, you can add the input fields and Submit button that will allow you to easily add new scores to the database.

Adding the Input Fields

To make life easy, we'll use Flash components for the input fields and Submit button. This will allow you to limit the amount, and type, of data entered into each field, as well as provide button rollover functionality without having to code anything.

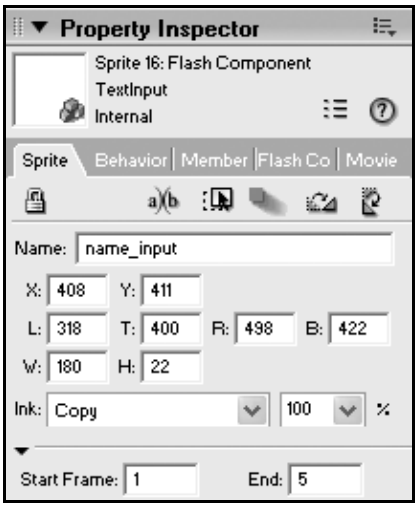
1. Change the toolset to Flash Component.



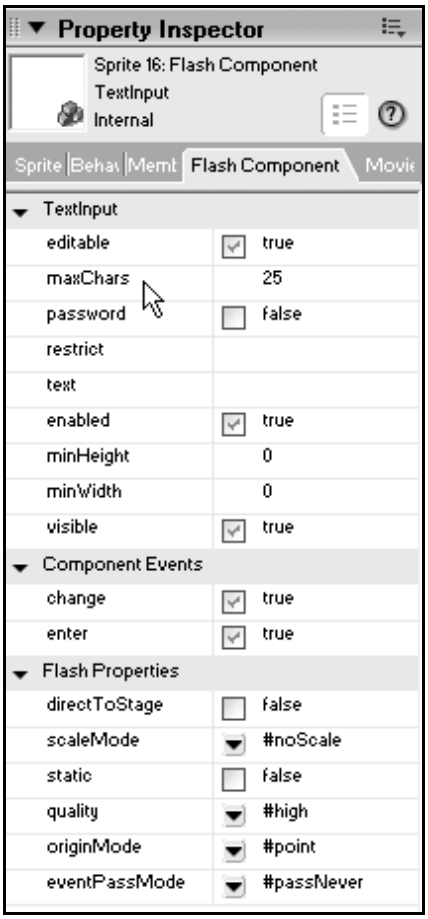
2. Select the TextInput tool and click on the Stage to create the sprite. Modify the sprite's span so it appears in channel 12 and occupies frames 1–5.

This will be the name input field.

3. Select the Sprite tab in the Property inspector. Name the sprite *name_input* and set its width to 180. Position the field underneath the score table.



4. Select the Flash Component tab in the Property inspector, and set the maxChars property to *25*. Leave the other settings at their default values.



By setting the maxChars property to 25, you limit the amount of text that can be entered to 25 characters, which matches the field length, for the name, you set up in the mySQL database.

5. Select the TextInput tool and click on the stage to create the score input field. Modify the sprite's span to occupy frames 1–5 of channel 13. Position the sprite on Stage, so it is next to the name input field.

Using a Flash component for the score entry will make it easy to limit the entered data to numbers only.

6. Select the Sprite tab in the Property inspector and name the sprite score_input. Choose the Flash Component tab. Set maxChars to *10*. Enter *0-9* into the restrict field.

Entering 0-9 into the restrict field for the component limits the entry to numbers only. Limiting the maximum number of characters to 10 will only somewhat limit the number you can input. Remember, the largest number that can be stored in the field is 2147483647, due to using an INT type for field in mySQL. This also matches Lingo's maxInteger value. However, you could defeat it by entering all 9s into the field, or actually any number larger than the maxInteger.

When you're creating a game, and not just a demo, you won't be allowing the player to enter their own score into the database. You will have to limit your scoring system so that the scores don't exceed the maxInteger value.

7. Select the Button component from the Tool Palette and click on the stage to create the button. Adjust its span so it occupies frames 1–5 of channel 14, then position the button near the input fields.

This will serve as the Submit button, allowing you to post the name and score entered into the fields to the database.

8. With the button selected, choose the Flash Component tab in the Property inspector. Enter *Submit* for the label. Leave the other properties set to their default values.

Now that you've got the interface completed, you can add the Lingo that will make it all work.

Creating the Behaviors

You've already done most of the work in creating the parent scripts. All that's needed now is to add a few basic behaviors that will instantiate the objects, and then check them for completeness.

1. Double-click the behavior channel at frame 1 to open a new behavior script window. Edit the exitFrame behavior to be as follows:

```
global hsDown

on exitFrame me
  hsDown = script("score_get").new()
  hsDown.setURL("http://www.myDomain.com/testing/scoreget.php")
  hsDown.downloadScores()
end
```

Placing the behavior at frame 1 will start the download of the high-score data as soon as the movie is played.

What you need to do now is add some code to the existing loop on frame behavior attached to frame 5 that will monitor the hsDown object to see when it's complete.

2. Double-click the frame behavior at frame 5 to open it for editing. Add the declaration for the hsDown global to the script.

```
global hsDown
```

Declaring the variable at the top of the script, before any handlers, makes hsDown available throughout the current script.

3. Add the following conditional test to the exitFrame handler. Add the test before the _movie.go(_movie.frame) line of code.

```
if objectP(hsDown) then
  if hsDown.isDone() then
    if hsDown.getErr() = "OK" then
      scoreList = hsDown.getHighScoreList()
      numItems = scoreList.count()
      scoreText = ""
      repeat with cnt = 1 to numItems
        thisName = scoreList[cnt][1]
        thisScore = scoreList[cnt][2]
        scoreText = scoreText & thisName & tab & thisScore & return
      end repeat
    end if
    member("score_display").text = scoreText
    hsDown = 0
  end if
end if
```

First, you test to see if hsDown is an object. If it is, you check to see if it is done and then test it further to see if the returned error code is OK. This should be quite familiar from your testing in the Message window.

Once the download has finished, and no error is detected, the getHighScoreList method is called, which places the returned list of high-score data into the scoreList variable. The number of items in the list is then placed into numItems and the scoreText variable is set to an empty string.

Next, a repeat loop iterates through the scoreList, pulling out each individual name and score and placing them into thisName and thisScore, respectively. The name and score are then appended to the end of the scoreText string. Note the use of the tab constant being used between the name and the score, as well as the return constant at the end; this is what forces the columns to line up properly and places each name and score on their own line. The following example shows what happens when scoreList contains three scores:

```
scoreList = [["Jackie",15000],["Jessey",12000],["Dave",10000]]
```

cnt	thisName	thisScore	scoreText
1	Jackie	15000	Jackie 15000
2	Jessey	12000	Jackie 15000 Jessey 12000
3	Dave	10000	Jackie 15000 Jessey 12000 Dave 10000

When finished, the text of member score_text is set to the scoreText string, causing it appear in the sprite on Stage. Notice at the end you're setting hsDown = 0. What this does is make the entire set of if statements stop executing, because the outer one that tests if hsDown is an object will now fail. It also destroys the object, removing it from memory. However, all you need to do to create the object again is to go to frame 1. This is what we'll do after submitting a score, so that the display is updated each time a new score is submitted.

4. Right-click the Submit button, select Script from the context menu, and create the following handler. Be sure and delete the default mouseUp.

```
global hsUp

on click me
  newName = sprite("name_input").text
  newScore = value(sprite("score_input").text)
  if newName <> "" and newScore <> 0 then
    hsUp = script("score_save").new()
    hsUp.setURL("http://www.myDomain.com/testing/scoresave.php")
    hsUp.postScore(newName, newScore)
  end if
end
```

Because the button is a Flash component, you need to use the on click method instead of the normal on mouseUp method. That way, when the button is clicked, the name and score are retrieved from the input fields and placed into newName and newScore. The value function is used to turn the score text into a regular number. Note that with Flash input fields you get the text through the sprite's text property. With standard Director text and field sprites, you get text through the member's text property.

A test is performed to make sure both a name and number have been entered into the fields. If they have, the global hsUp object is created from the score_save script. The URL to the PHP script is set and then the postScore method is called and sent the new name and new score to be posted into the database.

Now you need to add another test to the loop on frame script in order to test for the hsUp object. When it's finished uploading, you'll send the playback head to frame 1, which will again initiate the hsDown object and retrieve the high scores, displaying the newly submitted score on screen.

5. Double-click the frame behavior at frame 5 and add the following code to the end of the behavior, immediately preceding the _movie.go(_movie.frame) line.

```
if objectP(hsUp) then
  if hsUp.isDone() then
    if hsUp.getErr() = "OK" then
      _movie.go(1)
    end if
    hsUp = 0
  end if
end if
```

Now, when a score is being submitted, and hsUp is an object, the code will execute and check to see when hsUp has finished uploading. When it has, and the error comes back as OK, the playhead is sent to frame 1 in order to download the new high-score data. Notice that hsUp is being set to 0 after the playhead is sent to frame 1. You might expect that once the playhead leaves the frame, the code will stop executing. But it doesn't. In fact, the code continues to execute until it reaches the end—even after the playhead has left the frame. Doing it this way ensures that hsUp is set to 0 when the submittal has finished, even if an error was reported.

Before closing the script window, you just need to declare hsUp to be a global variable.

6. Add *hsUp* to the global declarations at the top of the script, and then close the Script window.

```
global hsDown, hsUp
```

You now have a working system that creates and destroys your child object as needed, allowing you to retrieve and submit data whenever you like. Before testing, you should add a simple behavior to the text sprite that erases any text in the member on beginSprite.

7. Right-click the text sprite on Stage, or in the Score, and select script from the context menu. Delete the default mouseUp handler and add the following:

```
on beginSprite me
  sprite(me.spriteNum).member.text = ""
end
```

8. Save the movie. Rewind and play it.

After a brief pause, the high-score data will appear. Go ahead and submit more names and scores to verify your code is working.

At this point you've got a demo movie, along with code, that you can use to add high-score functionality to any project you like.

Let's now discuss limiting the amount of data you are storing in the table.

Limiting the Data

Currently there are no checks in place that will limit the amount of data stored in the table. Each new high score is simply added to the table without regard for the table's size. Because your text field only displays 15 high scores, there's really no reason to store more than 15 names and scores in the database. Along these same lines you'll also want to perform a check to make sure a new score being submitted is high enough to be added to the database.

You can accomplish this by adding a new global: lowScore. When there are fewer than 15 scores in the database, lowScore will be 0. When there are 15 or more scores, you set lowScore to the value of score number 15. Then, as long as the new score being submitted is larger than lowScore, you can add the new score.

Taking this a step further, you can send lowScore to the PHP save script and have it delete any names and scores whose score is lower than lowScore. This will keep your table from getting too large by limiting the amount of data to just 15 scores.

1. Double-click the frame behavior at frame 5 and add the following immediately preceding the start of the repeat loop:

```
lowScore = 0
if numItems >= 15 then
  numItems = 15
  lowScore = scoreList[numItems][2]
end if
```

This way, lowScore will be 0 unless there are 15 or more scores in the table, in which case lowScore will be set to the value of the 15th score.

2. Add the global declaration for lowScore to the very top of the script, then close the script window.

```
global lowScore
```

Now you need to modify the behavior attached to the Submit button so that only a large enough score is submitted.

3. Right-click the Submit button and select Script from the context menu. Modify the behavior so it appears like the following:

```
global hsUp, lowScore

on click me
  newName = sprite("name_input").text
  newScore = value(sprite("score_input").text)
  if newName <> "" and newScore > lowScore then
    hsUp = script("score_save").new()
    hsUp.setURL("http://www.myDomain.com/testing/scoresave.php")
    hsUp.postScore(newName, newScore, lowScore)
  else
    alert "Score to low!"
  end if
end
```

Now, instead of checking if newScore is greater than 0, you check to see if it's larger than lowScore. If it is, you call the postScore method of the child object as before, except that now you send lowScore as a new parameter. Also notice that an alert has been added that will tell you when the submitted score is too low.

You can now modify the postScore method of the score_save parent script to accept the lowScore variable.

4. Double-click the score_save script in the cast to open it for editing. Modify the postScore method so it appears like the following:

```
on postScore me, nam, scor, lowScor
  myDone = False
  myErr = 0
  myNetID = postNetText(myURL, ["name":nam, "score":scor, "low":lowScor])
  _movie.actorList.add(me)
end
```

All you're doing here is sending the low score to the PHP script, which you will need to modify next.

5. Close the script window and save the movie. Launch your HTML editor and open the scoresave.php script. Modify it as follows:

First, add the following line to the top of the script, immediately following the other two lines that retrieve the name and score:

```
$lowscore = $_POST['low'];
```

Add the following lines immediately preceding the very last line that exits the script with a "1":

```
$sql = "DELETE FROM scores WHERE score < $lowscore";
$result = mysql_query($sql, $link);
if (mysql_errno()){
  exit("-5");
}
```

What this will do is remove all data from the scores table where values in the score field are lower than the low score value. If any error occurs a "-5" will be returned. If no error occurs, the last line in the script executes and returns "1", indicating success.

6. Save the PHP script and upload it to the server. Return to Director and play the movie. Submit new names and scores so that you have more than 15 of them in all.

If you'd like to verify the modifications are working properly, you can add a trace command to the frame behavior, at frame 5, to output the scoreList variable after calling the getHighScoreList method of the hsDown object.

7. When you're finished testing, stop the movie and save it.

Before finishing this lesson. let's discuss some simple security measures you can put in place to stop people from submitting score data on their own.

Adding a Level of Security

In this section we'll discuss some basic security measures you can put in place to stop would-be hackers accessing your high-score data. This could be necessary later, depending on the type of game you're building or the data you're storing. Many online games give out prizes for top scorers. For these types of games you will definitely need to add some security measures because people may try to cheat.

With your PHP scripts on the server all someone would need to do is know the URL and the data to be submitted. With some trial and error and simple guessing, a hacker could easily send new high scores to your database.

The first step in combating this would be to remove the user name and password from the PHP scripts and instead send them in from Director, like any other variable. This measure will actually go a long way to securing your data—as long as you never give out the user name and password.

If that's not enough, you can add a simple encryption algorithm that creates a key from your data. The key is stored in the database, and analyzed when the data is retrieved. Have a look at the following encryption method:

```
on encrypt nam, scor
  stringToEncode = nam & string(scor)
  enc = ""
  repeat with cnt = 1 to stringToEncode.length
    thisChar = stringToEncode.char[cnt]
    thisChar = bitXOR(charToNum(thisChar),3)
    thisChar = bitXOR(thisChar,15)
    enc = enc & numToChar(thisChar)
  end repeat
  return enc
end
```

This takes a name and a score and appends them together into one string, stringToEncode. A repeat loop pulls out each character from stringToEncode, turns it into a number (using charToNum), then uses Lingo's bitXOR method twice to modify the number. The modified number is turned back into a character and appended to the enc variable, which is returned when the repeat loop finishes.

Have a look at the following example:

```
encd = encrypt("Dave", 15000)
trace(encd)
-- "Hmzi=9<<<"
```

You can then store the encrypted string in the database, in its own field, along with the name and score. When you retrieve the data out of the database you run the encryption in reverse order to obtain the original string from the key:

```
on decrypt theKey
  dec = ""
  repeat with cnt = 1 to theKey.length
    thisChar = theKey.char[cnt]
    thisChar = bitXOR(charToNum(thisChar),15)
    thisChar = bitXOR(thisChar,3)
    dec = dec & numToChar(thisChar)
  end repeat
  return dec
end
```

Note how you do the bitXOR with 15 first, and then 3—the opposite of what you did when encrypting the data. Let's see how it works:

```
trace(decrypt("Hmzi=9<<<"))
-- "Dave15000"
```

Perfect! Unless someone knows the algorithm you used to create the key, they will have a very difficult time placing the correct data into the key field of the database.

You could even take this further and only store the encrypted data, making it that much more difficult to break into. There are also Xtras available that use much more sophisticated encryption algorithms that will guarantee your data can't be hacked.

For some great information on encryption, along with a purely Lingo implementation of the very secure blowfish algorithm, I suggest you visit Robert Tweed's Killing Moon site at:
http://www.killingmoon.com/director/lingofish/

What You Have Learned

In this lesson you have

- Created a mySQL database on your Web server (pages 3–8)
- Created PHP scripts to interface to the database (pages 8–16)
- Written parent scripts that use netLingo to talk to the PHP scripts (pages 16–25)
- Used SQL to add, retrieve, and delete data from the database table (pages 25–34)
- Learned to limit the amount of data being stored and retrieved (pages 34–36)
- Learned about basic security measures to secure your data (pages 36–37)