# Chapter 22

# Creating Your Own Objects and Behaviors

In the previous chapter, you learned how to take advantage of Dreamweaver extensions written by other people. Now it's time to learn to write your own! In this chapter,

you'll get some hands-on experience creating custom objects and behaviors, and you'll learn how to package them up and share them with the rest of the Dreamweaver community. In particular, you'll learn to do the following:

- Create a simple object that inserts the same code into the page every time
- Create a custom object that uses a dialog box to collect user input and insert customized code into the page
- Create an editable, reusable custom behavior
- Troubleshoot and bulletproof objects and behaviors for distribution to others
- Package a behavior or object into an installation file that can be used with Extension Manager
- Submit a packaged extension to the Macromedia Exchange

## Before Getting Started

Although writing Dreamweaver extensions isn't just for propeller-heads, it isn't for sissies, either. To work with object and behavior files, you need to be fairly comfortable reading and writing HTML code, and you need at least a basic understanding of JavaScript. In particular, before you tackle this chapter, you should be familiar with the following:

- The basic language structure, syntax requirements, and concepts of JavaScript (expressions, operators, variables, and so on)
- How to work with JavaScript functions
- How to use JavaScript to process data collected by HTML forms

**Tip**

If you're a JavaScript newbie, or if your skills are rusty, you might want to have a JavaScript reference available as you work. The handiest reference is Dreamweaver's own online JavaScript Reference panel. If you want more in-depth information, the O'Reilly series books on JavaScript (*JavaScript: The Definitive Guide*, *JavaScript Pocket Reference*) are a valuable resource. Danny Goodman's *JavaScript Bible* is also a wonderful reference and teaching resource.

In addition to the standard rules and regulations of JavaScript, Dreamweaver has its own application program interface (API), consisting of predefined objects, functions, and procedures for processing scripts. In the course of this chapter, you'll get a taste of the Dreamweaver API, and you'll be introduced to the parts of the API that you'll need

to write basic objects and behaviors. If you want to go beyond this chapter and seriously explore Dreamweaver extensions, the best resource is Macromedia's own manual, *Extending Dreamweaver*. This manual comes in PDF format on the Dreamweaver application CD; it can also be downloaded from the Dreamweaver support page on the Macromedia Web site (`www.macromedia.com/support/dreamweaver`). I highly recommend printing it out, popping it in a three-ring binder and keeping it next to your pillow—er, your computer—as you work.

So, are you ready to start extending?

## Working with Objects

In the last chapter, you learned that objects are created from files stored in Dreamweaver's Configuration/Objects folder. In this section, you'll learn what a well-formed object file looks like, how Dreamweaver processes object files, and how to build your own objects from scratch.

### What Are Objects?

An object, in terms of Dreamweaver's API, is an HTML file that contains, or uses JavaScript to construct a string of HTML code to insert into the user's document. That string of code can be anything from this:

```
copyright John Smith, 2000
```

to this:

```
<font face= "Georgia, Times, Times New Roman, serif" size="2">copyright
John Smith, 2000</font>
```

to this:

```
<table width="200" height="200" border="1">
       <tr>
              <td align="center" ><font face= "Georgia, Times, Times New
Roman, serif" size="2">copyright John Smith, 2000</font></td>
       </tr>
       </table>
```

In other words, it's anything that can validly sit in an HTML document. The code gets inserted wherever the user's cursor is when the object is chosen.

#### *What Files, Where*

If you look inside the Configuration/Objects folder, you'll see several subfolders, the names of which you should recognize as the categories of objects accessible in the

Objects panel. Open up one of those folders (Common, for instance), and you'll see dozens of files, corresponding to the individual objects in the Common Objects panel. Each object consists of from one to three files, all with the same name but different extensions. These files are listed here:

- **An HTML file** (Table.html, for instance). This is the object file itself, the file that contains or returns the code to be inserted. This is the only file that *must* be present to constitute an object.
- **A JavaScript file** (Table.js, for instance). This file contains JavaScript instructions for constructing the code to be inserted, in the form of one or more JavaScript functions, and is called on from the HTML file. This file is optional: It is entirely legal to contain the JavaScript functions in the head section of the object's HTML file instead of saving it to an external file. As experienced scripters know, it can be easier to keep track of and update JavaScripts if the code is in a separate file, but it isn't necessary.
- **An image file** (Table.gif, for instance). This file contains a 16×16 pixel image that Dreamweaver uses to represent the object in the Objects panel. This file is also optional: If there is no image file, Dreamweaver will supply a generic image icon to represent the object in the panel.

Figure 22.1 shows some typical sets of object files.

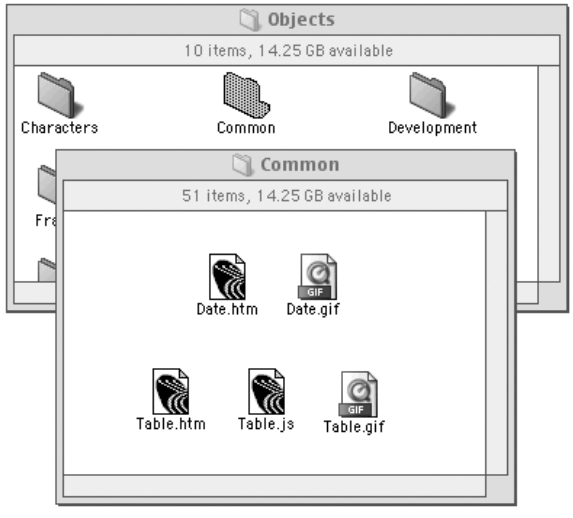


**Figure 22.1** The Objects/Common folder, containing the files for the Insert Date object (two files only) and the Insert Table object (three files).

Object files must be stored in one of the folders inside the Configuration/Objects folder. The folder in which they're stored determines what portion of the Objects panel they'll appear in.

### *Structure of a Simple Object File: No Dialog Box*

Some objects use dialog boxes to collect user information, and some don't. Those that don't are (not surprisingly) easier to create. Figure 22.2 shows a simple object file that doesn't call a dialog box.

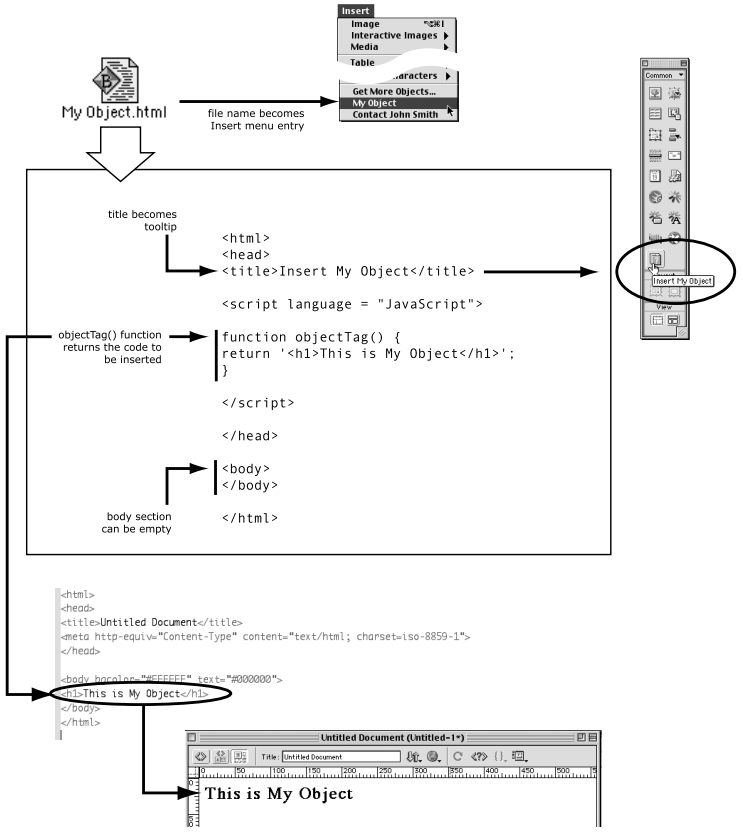


**Figure 22.2** The structure of a simple object file, and how it translates into a Dreamweaver object.

The key elements of the file are as follows:

- **Filename**. This becomes the Insert menu entry for the object.

- **Page title**. This becomes the ToolTip that pops up to identify the object in the Objects panel.
- **objectTag() function**. This JavaScript function is the most important element of the object file. The function returns the exact code that you want the object to insert into your document, enclosed in quotes. The objectTag() function is part of the Dreamweaver API, so it doesn't need to be defined. It also doesn't need to be called; Dreamweaver calls it automatically when the object is chosen.

In the example shown in Figure 22.2, the code returned by the objectTag() function is a simple level 1 heading. Notice how everything between the quote marks, in the return statement, becomes part of the user's document.

### *Structure of a Fancier Object File with a Dialog Box*

More sophisticated objects do more than insert pre-established code; they collect user information and use that information to customize the inserted code. Figure 22.3 shows the structure of an object file that creates a dialog box to collect user input and then constructs the code to insert based on that input.

The added element in this kind of object file is HTML Form, which becomes the dialog box for collecting user input and customizing the code. Note that the form as written doesn't include a Submit button. Dreamweaver supplies the OK and Cancel buttons automatically. The form needs to have only the fields for collecting data.

In the example shown in Figure 22.3, the code returned by the objectTag() function is a level 1 heading, but with the content to be determined by the user. Notice how a variable is used in the objectTag() function to collect the form input and pass it along to the return statement.

## Making Your Own Objects

Did the previous information seem awfully theoretical? In this section, you'll get some hands-on experience making your own objects. You'll move from simple objects to complex, and along the way you'll learn some tips and tricks for efficient object-handling.

### *Before Getting Started…*

Writing your own objects involves hand coding, for which you'll need a text editor. If you're an experienced scripter, you probably already have a favorite text editor (NotePad, SimpleText, BBEdit, HomeSite, and so on); you can use any text editor to code your Dreamweaver objects. You are also free to use Dreamweaver itself to code your objects;

both Layout view and Code view come in handy. Although it might seem like a dangerous way to proceed, Dreamweaver has no problem editing its own extensions. This is because Dreamweaver accesses only the extension files when it's loading extensions, which happens only when the program launches or when you force it to reload extensions.

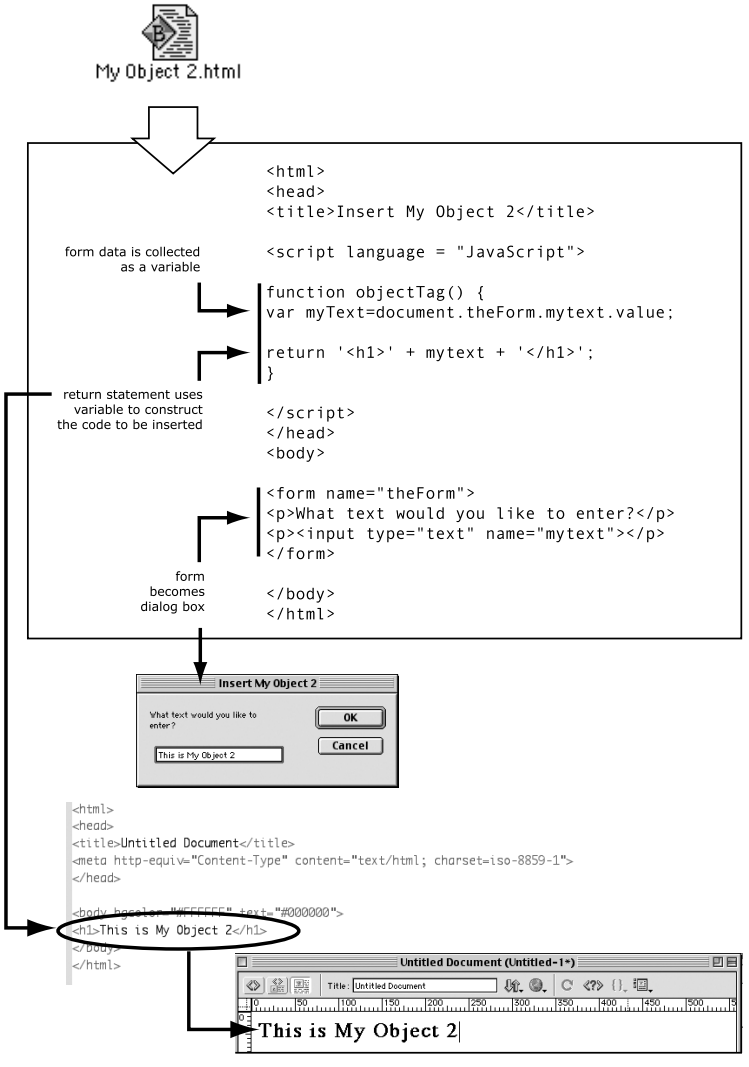


**Figure 22.3** The structure of a full-featured object file, and how it translates into a Dreamweaver object.

When you're coding and testing extensions, it's easy for file management to get out of hand. A good strategy is to create a special working folder, stored outside the Dreamweaver application folder. Keep the following items in this folder:

- A backup copy of the Configuration folder. Whether you're adding new extensions or changing existing files, it's courting disaster to work on your only copy of this folder. If things get hopelessly messed up, you can replace your customized Configuration folder with this clean one; it's quicker and easier than reinstalling the whole program.
- A shortcut (PC) or alias (Mac) to Dreamweaver's Configuration folder and any of its subfolders you'll be accessing frequently.
- Test files (you'll be generating several of those in this chapter's exercises).
- Files for packaging extensions.
- Any extensions that you want to keep inactive during your development work.

While you're working on the exercises in this chapter, keep your working folder open, or leave a shortcut or alias to it on your desktop; otherwise, you'll be doing a lot of navigating through your hard drive.

### Exercise 22.1 Setting Up Shop

In this exercise, you'll get your working space in order and learn some of the basic extension-developing features available in Dreamweaver. You'll create a custom objects folder where you can store your exercise objects, and you'll get some practice loading and reloading extensions.

1. Make sure Dreamweaver isn't running. If Dreamweaver is open and running on your computer, quit the program. You do this to experiment with how and when Dreamweaver loads extensions.
2. Find and open the Configuration/Objects folder on your hard drive. As you saw previously, every object file must be stored in a folder within the Configuration/Objects folder; every one of those folders correlates to a set of objects in the Objects panel. You can also add new object sets to the panel by adding your own new folders to the Configuration/Objects folder.
3. Create a new folder inside the Configuration/Objects folder. Name it Custom. When you're developing new objects, it's a good strategy to put them in a special folder called Custom or Development, at least until you're sure they're running properly.
4. Launch Dreamweaver. Every time you launch Dreamweaver, the program checks the Configuration folder and loads all extensions inside.

5. Check the Objects panel for a new category called Custom. In the Objects panel, click the triangle that accesses the pop-up Objects menu. There should now be a category called Custom (see Figure 22.4). Of course, if you choose that category, it'll be empty; that's because, so far, the Custom folder that you created is still empty.

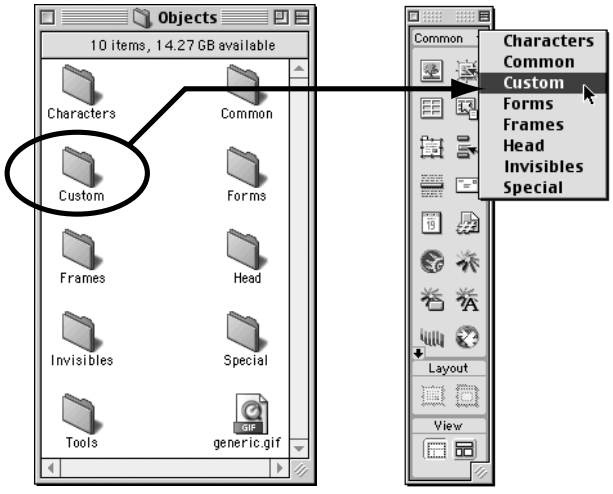


**Figure 22.4** New Custom folder in the Configuration/Objects folder, and the resulting Custom category in the Objects panel.

6. Without quitting Dreamweaver, rename the Custom folder as Development. If you're on a PC, minimize Dreamweaver; if you're on a Mac, hide Dreamweaver or send it to the background. On your hard drive, go back to the Configuration/Objects folder and find your Custom folder. Rename it Development.
7. In Dreamweaver, check the Objects panel categories again. Go back to Dreamweaver. Repeat step 5, checking the Objects panel to see what categories of objects you have to choose from. Your new category should still appear as Custom, not Development. This is because Dreamweaver hasn't yet noticed the name change.
8. Force Dreamweaver to reload extensions without quitting and relaunching. Hold down the Ctrl/Opt key and click the pop-up triangle in the Objects panel. The Reload Extensions command should now appear at the bottom of the pop-up menu (see Figure 22.5). Choose that command. Although some extension-related tasks require the program to be relaunched, most updating can be done more quickly this way.

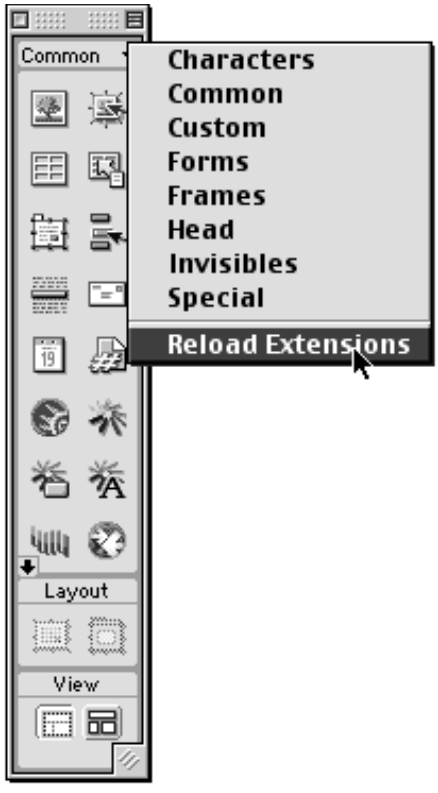


**Figure 22.5** Ctrl/Opt-clicking the pop-up triangle in the Objects panel reveals the Reload Extensions command.

9. Check the Objects panel categories again. Now release the Ctrl/Opt key and click the pop-up triangle to access the object categories. Your custom category should now show up as Development.

   What does this mean? Dreamweaver doesn't constantly access the object files; rather, it accesses them once as it launches. Any time you change an object file, you must make Dreamweaver notice your changes by either quitting and relaunching the program or Ctrl/Opt-clicking the Objects panel pop-up and choosing Reload Extensions.

## Exercise 22.2 Making a Simple Object

The simplest objects are those that don't call up a dialog box for user input and, therefore, always return the same code. The simple object that we'll create in this exercise is a contact information line that links to an email address—just the sort of thing you might want to put at the bottom of all your Web pages.

To create this object, use the text editor of your choice. Save all exercise files in the Development folder that you created in the last exercise.

1. Decide on the exact line of code that you want the object to insert. In this case, this will be a one-line piece of text, formatted however you like, with an email address embedded in it. The result should look like Figure 22.6.

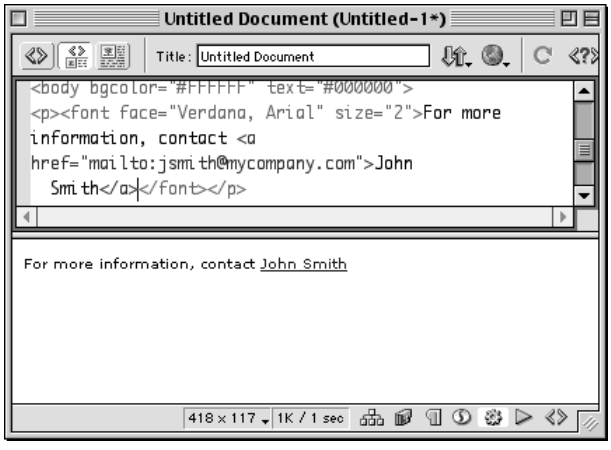


**Figure 22.6** The inserted code for the Insert John Smith Contact Info object (shown in Layout and Code view).

If you don't want to type in all this code by hand, remember that you can use Dreamweaver's visual editor to create the final result; then go to the Code Inspector or Code view, and the code will be there, written for you.

2. Create the basic object file, with all structural elements in place. Open your text editor—or, if you prefer to work in Dreamweaver, open the Code Inspector—and enter the basic required code as described previously. You can leave out the details specific to this object, for now. Your code framework should look like this (elements that you'll be replacing later with custom text appear in bold):

```
<html>
<head>
<title>Title Goes Here</title>
<script language="JavaScript">
function objectTag() {
return 'inserted code goes here';
}
</script>
</head>
<body>
</body>
</html>
```

**Tip**

If you write a lot of objects, you might get tired of writing this code framework over and over. You can save yourself all that typing by saving this empty framework as a text file and storing it in your working folder or in some other easily accessible spot.

3. Enter a page title into the code. This will become the ToolTip that shows in the Objects panel.

   Most Dreamweaver ToolTips start with Insert (though this is convention only, not a requirement). A logical title for the current file, therefore, might be Insert John Smith Contact Info. The top portion of your code should now look like this:

```
<html>
<head>
<title>Insert John Smith Contact Info</title>
```

4. Insert the desired line of code as the return statement of the objectTag() function. If you have the line of code already typed into your computer, you can just copy and paste it in; otherwise, type it in manually now.

   Note that the entire return statement must be in quotes. They can be single or double quotes; just make sure that they're in balanced pairs.

   Your code should now look like this:

```
<html>
<head>
<title> Insert John Smith Contact Info </title>
<script language="JavaScript">
function objectTag() {
return '<p><font face="Verdana, Arial" size="2">For more informa-
tion,    contact    <a    href="mailto:jsmith@mycompany.com">John
Smith</a></font></p>';
}
</script>
</head>
<body>
</body>
</html>
```

**Note**

As with any JavaScript return statement, there can be no hard returns within the statement. Make sure that your code is written out in one long line, or it won't work!

5. Save your file in the Development folder. Call it Contact John Smith.html.

   Remember, object files must be in the proper folder, or they won't be recognized as objects. Make sure that you're saving the file into your Development folder. The extension can be .htm or .html—Dreamweaver accepts either.

The filename will become the menu command that appears in the Insert menu, so it's good practice to name it something descriptive. Unlike browsers and Web servers, Dreamweaver allows spaces in filenames and respects case changes. A filename such as Contact John Smith.html will work fine and will look good in the Insert menu. A filename such as jsmith_contact_info_1.html will also work, but it won't look as nice (or be as understandable) in the menu. Capitalization will also carry through to the menu entry.

6. Test your object!

   If you already have Dreamweaver running, Ctrl/Opt-click the pop-up triangle in the Objects panel to access the Reload Extensions command.

   If you don't have Dreamweaver running, launch it—the extension will be loaded automatically.

   Create a new document, if there isn't one already open.

   Check the Development category of the Objects panel. The new object should be there, represented by a generic object icon. Position the cursor over the icon, and the ToolTip should appear (see Figure 22.7).

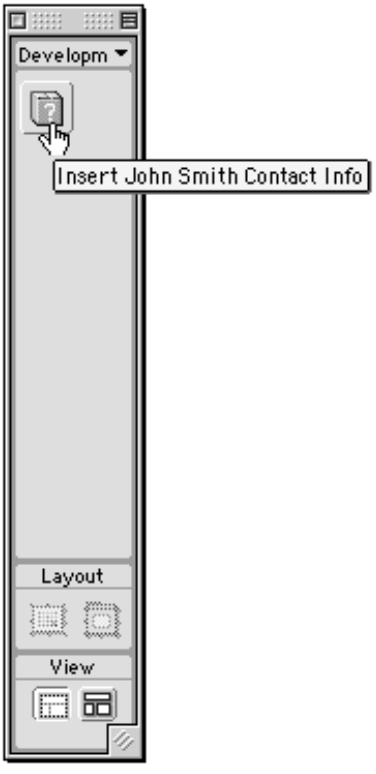


**Figure 22.7**
The new custom object with a generic icon and ToolTip.

Click on the object. Your desired code should be inserted into the document.

Congratulations! You've made your very first object.

**Tip**

Don't waste your time making custom icon files for objects while they're still in the development phase. Wait until the object is all polished and perfectly functioning; then dress it up with a custom icon. (See Exercise 22.6 for how to make an icon file.)

**Note**

If your object doesn't show up in the Objects panel at all, then either you saved it in the wrong place, you didn't append the HTML extension to the filename, or you need to reload Dreamweaver's extensions. Even an invalid object file will show up in the Objects panel if it has a valid name and is stored in a valid location.

If your object shows up but there's something wrong with the code, you'll probably get an error message when Dreamweaver tries to execute the objectTag() function. Dreamweaver's error messages are fairly specific in what went wrong and what needs fixing.

### Exercise 22.3 Adding a Dialog Box

Your simple object is fine as far as it goes, but it's not a very flexible or useful object because it always returns the same code, no matter what. What if you want to link the contact information to someone other than good old John Smith? A fully functional object would bring up a dialog box that would ask for user input and then would enter that information into the code. That's the object you'll build next.

1. Open Contact John Smith.html and save it as Contact Info.html. Why reinvent the wheel? Let's build on our previous success by adding a dialog box and tweaking the objectTag() function's return statement to collect user input.
2. Change the page title of the new object file to Insert Contact Info.

   Remember, the page title creates the ToolTip that identifies the object in the Objects panel; right now, the page title of this file is identical to the title of the object that you worked on earlier. This makes it difficult to tell the objects apart in the panel, especially because both objects currently have generic icons.
3. Decide what pieces of the code you want to replace with user input. Check Figure 22.6 to remind yourself what the end product should look like.

   For this object, you'll probably want to ask the user for a contact name (instead of John Smith) and an email address (instead of sending everything to poor John Smith's email).
4. Create an HTML form that will serve as a dialog box to collect this information. To be functional, your form will need two text fields: one to collect the name and another to collect the email address. So, the simplest form you could possibly come up with might look something like the one shown in Figure 22.8.

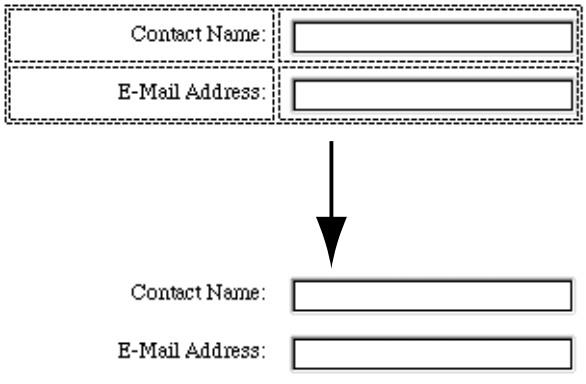


**Figure 22.8** The form for the Contact Info object dialog box, as it appears in Dreamweaver's Layout view and as you want it to finally appear.

Open Contact Info.html and build the form in the body section of that file. If you like coding forms by hand, go to it. If you'd rather use a visual editor, open

the file in Dreamweaver and use Dreamweaver's visual editor to build it. Figure 22.8 shows how the designed form might appear in Dreamweaver's Layout view.

**Note**

If you're building your form in Dreamweaver's visual editor, method and action properties will be automatically added to the <form> tag. Your form doesn't need either of those because the form isn't going to be processed in the standard way. You can safely remove these properties from your code. Dreamweaver will also add background and text color properties to the <body> tag; you should remove these and let Dreamweaver determine the appropriate color scheme for the dialog box.

Your form code should look like this:

```
<form name="theForm">
<table>
       <tr valign="baseline">
              <td align="right" nowrap>Contact Name:</td>
              <td align="left">
                     <input type="text" name="contact" size="30">
              </td>
       </tr>
       <tr valign="baseline">
              <td align="right" nowrap>E-Mail Address:</td>
              <td align="left">
                     <input type="text" name="email" size="30">
              </td>
       </tr>
</table>
</form>
```

5. Add variables to the objectTag() function to collect form data.

   Open the Contact Info.html file in your text editor. The first step in processing user input is to declare two variables to collect the information from the two text fields in the form. These variables will be local to the objectTag() function and thus must be declared inside it.

   Add two local variable declarations to your function, each initialized to the value of one of the form fields. The code of your objectTag() function should now look like this:

   ```
   function objectTag() {
   var contact = document.theForm.contact.value;
   var email = document.theForm.email.value;
   return '<p><font face="Verdana, Arial" size="2">For more informa-
   tion,    contact    <a    href="mailto:jsmith@mycompany.com">John
   Smith</a></font></p>';
   }
   ```

6. Rewrite the return statement, substituting the two variables for the name and email address. If you're an old coding hand, this will be a piece of cake. If you're a novice at JavaScript, the trickiest bit is balancing the opening and closing quotes

so that you don't end up with any unterminated string literals.

Your objectTag() function should now look like this:

```
function objectTag() {
var contact=document.theForm.contact.value;
var email=document.theForm.email.value;
return '<p><font face="Verdana, Arial, sans-serif" size="2"> For
more information contact <a href="mailto:' + email + '">' + contact
+ '</a></font></p>';
}
```

**Note**

Why declare variables instead of putting the references to the form directly into the return statement? No reason, except that it keeps the return statement from getting too long and unwieldy. The following code would work just as well:

```
return '<p><font face="Verdana, Arial, sans-serif" size="2"> For more
information contact <a href="mailto:' + document.theForm.email.value
+ '">' + document.theForm.contact.value + '</a></font></p>'
```

Figure 22.9 shows your complete object file code.

File Edit Search Help

```
<html>
<head>
<title>Insert Contact Info</title>
<script language = "JavaScript">
function objectTag() {
var contact=document.theForm.contact.value;
var email=document.theForm.email.value;
return '<p><font face="Verdana, Arial, sans-serif" size="2"> For more information contact <a
href="mailto:' + email + '">' + contact + '</a></font></p>';
}
</script>
</head>
<body onLoad="initializeUI()">

<form name="theForm">
<table>
  <tr valign="baseline">
    <td align="right" nowrap>Contact Name:</td>
    <td align="left">
      <input type="text" name="contact" size="30">
    </td>
  </tr>
  <tr valign="baseline">
    <td align="right" nowrap>E-Mail Address:</td>
    <td align="left">
      <input type="text" name="email" size="30">
    </td>
  </tr>
</table>
</form>
</body>
</html>
```

**Figure 22.9** Complete code for the Contact Info.html object file.

7. Test your object. In Dreamweaver, reload extensions and try to insert the new object. You should get a lovely dialog box that looks like the one shown in Figure 22.10.

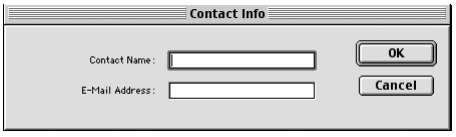


**Figure 22.10** Dialog box for the Contact Info object.

When you fill in your information and click OK, a customized contact information line should appear in your document.

**Note**

As with the previous exercise, if there's a problem with your code, Dreamweaver should give you a helpful error message. Read the error message, try to guess what it means, and then go back to your code and look for problems. Compare your code to the previous code to see what might be wrong.

The most common things that go wrong in this kind of object file are misnamed variables and form elements; invalid variable declaration and initialization statements; and mismatched single and double quotes in the return statement.

## Exercise 22.4 Refining Your Object

In this exercise, you'll see that, while the only required JavaScript function for an object is the objectTag() function, you can add other optional functions. In fact, you can define any function that you like in the head section of the object file. As long as you call the function in the <body> tag, using the onLoad event handler, Dreamweaver will execute the function as soon as the user chooses the object.

The local function that you'll add in this exercise addresses a minor annoyance that you may have noticed in the dialog box that your object calls up. When the dialog box comes up, the insertion point is not in the correct position for you to immediately start entering data. That's not a life-threatening problem, but it's less than slick.

1. Open the Contact Info.html file in your text editor. Because this is not major surgery, you'll work on the same object that you created in the last exercise instead of creating a new object.
2. Add an initializing function to the document head. Somewhere inside the <script> tags in the document's head section, add the following code:

```
function initializeUI()
{
document.theForm.contact.focus();
document.theForm.contact.select();
}
```

What does this function do? The first line officially gives focus to whatever form element is named within it—in this case, the Contact field (your first text field).

The second line selects the text (if any) in whatever form element is named within it—again, in this case, the Contact field.

**Note**

This function is used in many of the objects that ship with Dreamweaver. (Macromedia does not prohibit borrowing pieces of code.) Because the function is not part of the API, there's nothing magic about its name. If you'd rather name it something different, feel free to do so.

3. Call the function from the <body> tag. Add the following code to the <body> tag of the object file:

   ```
   <body onLoad="initializeUI()">
   ```

   Because this function is not part of Dreamweaver's official object-handling procedure, it must be specifically called.

4. Reload extensions and test the object. Now try it out. Save and close the object file. In Dreamweaver, reload extensions and choose the object from the Objects panel. The insertion point should be primed and ready to enter data into the Contact field of the dialog box.

## Exercise 22.5 Creating a Separate JS File

Although it won't affect the functionality of the object, you might decide that separating the JavaScript from the HTML portion of the object makes upkeep easier.

In this exercise, you'll separate the JavaScript and place it in a JS file, using a link inside the main object file (the HTML file) to access it.

1. Make a copy of the object file. Because this is a major change to your object, it's a good idea to make a copy of the original file to work on. You can use the File/Save As command in your text editor, or you can duplicate the file from Windows Explorer or the Macintosh Finder, whichever you prefer. If you keep both your duplicate and your original in the Development folder, you'll need to give each file a different name; if you want it to retain the same filenames, you'll have to move the original file to another folder. (Also, if you keep both files in the Development folder, Dreamweaver will show two generic icons in the Objects panel. If this is the case, you'll need to give each different object file its own unique page title so that the ToolTips that identify them in the Objects panel will be unique.)
2. Open the object file in your text editor or in Dreamweaver, and select the JavaScript in the head section.

   As you know if you're an experienced scripter, the JavaScript code that can be copied to an external file is the code that would normally appear in the HTML

document's head section—the functions, in other words. So you start by selecting all the code that appears between the opening and closing <script> tags. You should select the bold code shown here:

```
<html>
<head>
<title>Contact Info</title>
<script language="JavaScript">
function objectTag() {
var contact=document.theForm.contact.value;
var email=document.theForm.email.value;
return '<p><font face="Verdana, Arial" size="2">For more informa-
tion, contact <a
href="mailto:'+email+'">'+contact+'</a></font></p>';
}
</script>
</head>
```

3. Cut the selected code, and paste it into a new text document. Cut that code to the Clipboard (Edit/Cut). If you're working in a text editor, create a new document and paste in the code (Edit/Paste). If you're working in Dreamweaver, create a new document; then open the Code Inspector for that new document, and select all the code in the source editor and delete it. Then paste in the new code.

**Note**

Dreamweaver automatically supplies the HTML framework for all new documents; if you don't need that framework, it is safe to select the code and delete it.

4. Save the new document as Contact Info.js. The new document, which contains only the JavaScript code from the head of the original object file, must be saved as a text file with the .js extension. It should be stored in the Development folder, and named to match the object file.

**Note**

Actually, the filename doesn't need to match the object file, and the script file doesn't need to be stored in exactly the same folder as the object file, as long as the correct name and relative URL are given when the two files are linked. However, it is recommended to keep to a naming convention like this because it makes file management much easier.

5. In the object file, change the <script> tags to link to the JS file. The head section of the original HTML object file should now look like this:

```
<head>
<title>Contact Info</title>
<script language="JavaScript">
</script>
</head>
```

Change the opening script tag so that it looks like this:

```
<head>
<title>Contact Info</title>
<script language="JavaScript" src = "Contact Info.js">
</script>
</head>
```

6. Save and close both files. Figures 22.11 and 22.12 show what the complete code should look like in the two files.

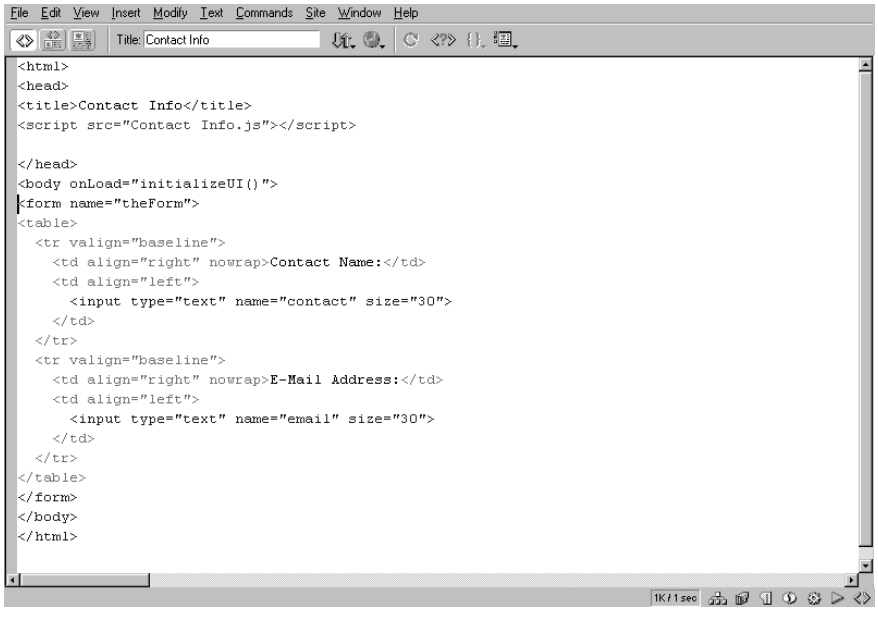


**Figure 22.11** The complete code for Contact Info.html, substituting the external file reference for the JavaScript.

7. In Dreamweaver, reload extensions and try out the object. If you correctly moved the JavaScript code and correctly entered the link information in the object file, the object should work in exactly the same way it worked before.

**Note**

If you linked from the object file to the JS file incorrectly, one of two things will happen: If there's a local function, such as initializeUI( ), Dreamweaver will report a JavaScript error when the <body> loads and tries to call that function (see Figure 22.13). If there are no local functions, Dreamweaver won't report an error, but the object will insert some strange code into your page. (The link from the HTML file to the JS file must be a relative address. If both files are in the same folder—which is definitely the safest way to go—the address will simply be the name of the JS file. If you want to store the JS file in another folder, you must enter the correct relative address, including any subfolder or parent folder information.)

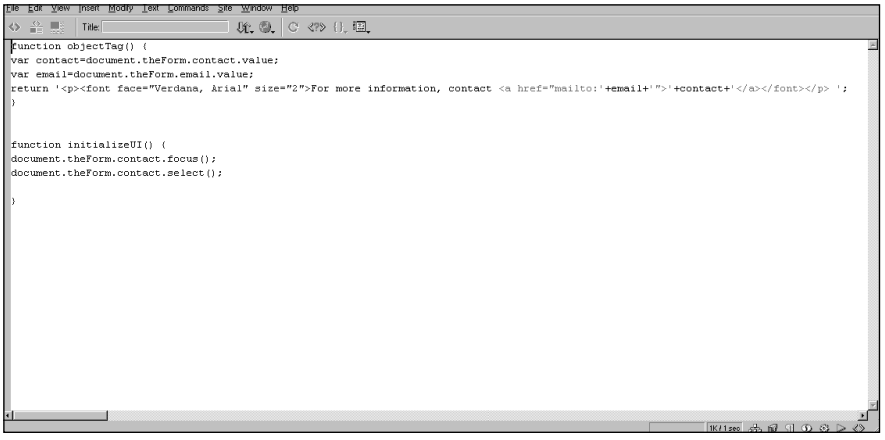


**Figure 22.12** The complete code for Contact Info.js.


**Figure 22.13** Dreamweaver error message showing the result of incorrect linking between the object file and its external JS file.

If you didn't paste the correct code into the JS file, you'll probably generate a JavaScript syntax error, and Dreamweaver will generate an error message.

## Exercise 22.6 Creating an Object Icon Using Fireworks

Professional-looking objects have their own icons. When the development phase of your object is done, the finishing touch is to make an icon file to represent it in the Objects panel.

The requirements for an icon file are as follows:

- The file must be a GIF image file, preferably no larger than 16×16 pixels. (Larger images will work, but they'll be squashed into an 16×16 pixel space in the panel.)
- The file must have exactly the same name as the object file it goes with, but with the GIF extension. For this exercise, therefore, the name must be Contact Info.gif.
- The file must be stored in the same folder as the object file it goes with. For this exercise, the icon file must be stored in the Development folder.

Icon files can have any colors you like, and the icon can look like anything you can imagine. You'll quickly discover, though, that designing icons that clearly communicate what they represent, when there are only 256 pixels to play with, is a real art.

This exercise won't show you how to create a graphic to use as an object icon; that's beyond the scope of this chapter. But you will see how to use an existing graphic as an icon file.

1. Create, adapt, or borrow a 16×16 pixel GIF file containing an icon. If you have access to a good graphics program (such as Macromedia Fireworks) and want to create your own icon, do it. Otherwise, use the Contact Info.gif file on the CD. (If you make your own file, make sure to name it Contact Info.gif.)

**Tip**

> You don't have to create your icons from scratch. You can start with an existing icon and adapt it to your needs. The Contact Info.gif file on the CD was adapted from Macromedia's E-Mail.gif file.

2. Put the icon file in the Development folder. Remember, the icon must be stored in the same folder as the object file.
3. Reload extensions, and take a look at your icon. In Dreamweaver, reload extensions.
4. In the Objects panel, access your Development objects. If all went as planned, you should have a beautiful custom icon in place (see Figure 22.14).

## Working Smart with Objects

Congratulations! You now know the foundation skills for making Dreamweaver objects. How can you make objects work for you?

### *Analyze Your Needs*

As you've seen, any piece of HTML code that you repeatedly use in Web pages is a candidate for an object. The best object candidates, though, are pieces of code that you need to customize and then insert—changing the name and email address, specifying a certain URL to link to, and so forth.


**Figure 22.14**
The Contact Info object, as it appears in the Objects panel with its new custom icon in place.

Any time you find yourself going through the same steps over and over as you add content to Web pages, ask yourself these questions:

- Is the code I'm inserting similar enough each time that I could create an object from it?

- Are there differences in the code each time, or is it exactly the same code? (If the code is exactly the same each time, requiring no customization, it might be more efficient to use a recorded command or even a library item.)
- How many more times do I think I'm likely to need to insert this code? Will my need continue after today, after the current assignment, indefinitely? Creating an object is a time-consuming solution if the need is only very temporary.
- Do I have some extra time right now to devote to making this object? (Never try a new, challenging solution when your deadline is 45 minutes away.)

Depending on your answers, you'll know if it's time to crack open Dreamweaver and fit a new custom object inside.

#### *Expand Your Horizons*

When you feel comfortable with the basic object-making framework as presented here, expand yourself. Read through the *Extending Dreamweaver* manual to get ideas of what's possible in the Dreamweaver API. Take a look at some of the objects that come with Dreamweaver, to see how they're structured and what refinements they include.

## Working with Behaviors

Ah, behaviors! This is where the glitz comes in, all those neat little prewritten JavaScripts that Dreamweaver allows you to add to your web pages at the click of a button. Believe it or not, basic behaviors are only slightly more complicated than objects to create—though, as always, the more you know about JavaScript, the better (and more spectacular) your results will be.

In this section, you'll learn what a well-formed behavior file looks like, how Dreamweaver processes behavior files, and how to build your own behaviors from scratch.

### What Are Behaviors?

A behavior, like an object, is a snippet of code that gets inserted into a Dreamweaver document when the user chooses from the behaviors list. Instead of adding HTML code to the document (also like an object), however, a behavior adds JavaScript code.

Behaviors are constructed the same way objects are: Dreamweaver API procedures and functions enable you to return the exact string of code that you want inserted into your document.

For a variety of reasons, however, behaviors are inherently more complex than objects:

- A behavior inserts two pieces of code—a function (in the document head) and a function call (in the body section, wherever the cursor is when the behavior is chosen).
- Preferably, a behavior should be editable after it has been inserted, by double-clicking it in the Behavior panel. (To edit objects after they've been inserted, you use the Property Inspector. Although it is possible to create a custom Property Inspector, it's not a task for the faint of heart.)
- Preferably, a behavior should specify which event handlers can be used in the function call; it should be inaccessible in the behaviors list (grayed out) if the current selection is inappropriate.

**Note**

Dreamweaver behaviors always insert JavaScript code into the document in the form of a generic function, defined in the head section, and a customized function call, defined in the body.

### *What Files, Where*

Like objects, each behavior consists of an HTML file (the behavior file), which either contains JavaScript code or is linked to an external JS file. Behavior files are stored in the Configuration/Behaviors/Actions folder. Like the Objects folder, the Actions folder can contain folders inside itself; each folder corresponds to a submenu within the behaviors list. Again like the Objects folder, any new folder added to the Actions folder will result in a new submenu in the behaviors list (see Figure 22.15).

### *Structure of a Simple Behavior File*

Like object files, behavior files have their own required structure, and Dreamweaver has a set procedure for dealing with them. Figure 22.16 shows the framework code of a basic behavior file, containing only the required elements.

The key elements of the file are listed here:

- **Page title.** This becomes the name of the behavior as it appears in the behaviors list.
- **Defined function.** This is the JavaScript function that the behavior will insert in the head section of the user's document. It will be inserted exactly as coded here. This function must have a unique name—no other defined function in the entire Actions folder can use this name.

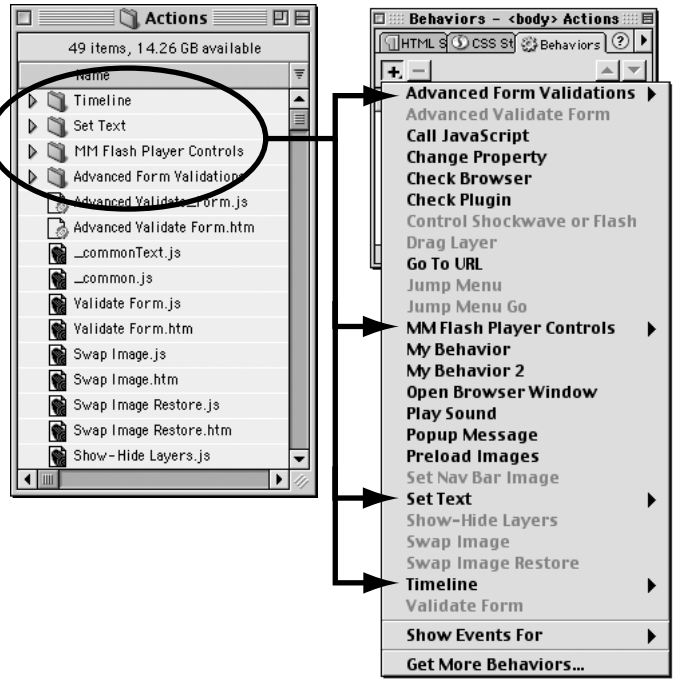


**Figure 22.15** The Actions folder, showing its subfolders, and the corresponding submenus in the Behavior panel's behavior list.

- **behaviorFunction() function.** This JavaScript function, part of the Dreamweaver API, must return the name of the function defined previously (without the ending parentheses). This function is called automatically when the behavior is chosen and, therefore, needn't be called in the behavior file.
- **applyBehavior() function.** This function, also part of the Dreamweaver API, must return the exact code of the function call as it will be inserted into the user's document. In a more complex behavior that requires arguments to be passed to the function, those arguments must be included in the return statement. This function is called automatically when the behavior is chosen and, therefore, needn't be called in the behavior file.
- **<body> tag elements.** Any code in the body section of the file will appear in a dialog box when the behavior is chosen. All behaviors automatically call up dialog boxes. If the behavior requires user input, create a form (as with object files). If the behavior requires no user input (as in the example here), some content must be placed in the body, or an empty dialog box will appear.

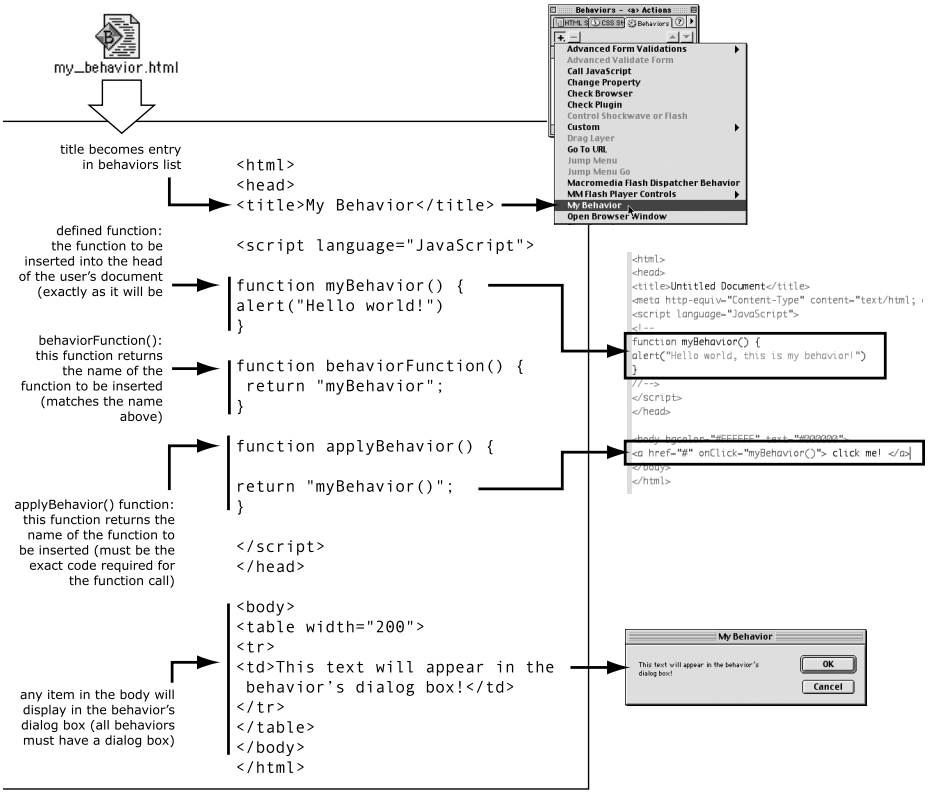


**Figure 22.16** The code framework required for a basic behavior file.

In the example shown in Figure 22.16, the function returned is a simple pop-up alert. Notice how the function and function call appear in the user's document, exactly as presented in the code.

### *Structure of a Fancier Behavior with a Dialog Box and Other Refinements*

The behavior shown previously contains all the necessary elements and will work properly. However, it is unusually simple (requiring no user input and passing no arguments from the function call to the function), and it is missing several key enhancements (such as control over the event handler used and features that will allow the behavior to be edited in the future). Figure 22.17 shows the code for a more full-featured behavior, complete with all these (optional) features.

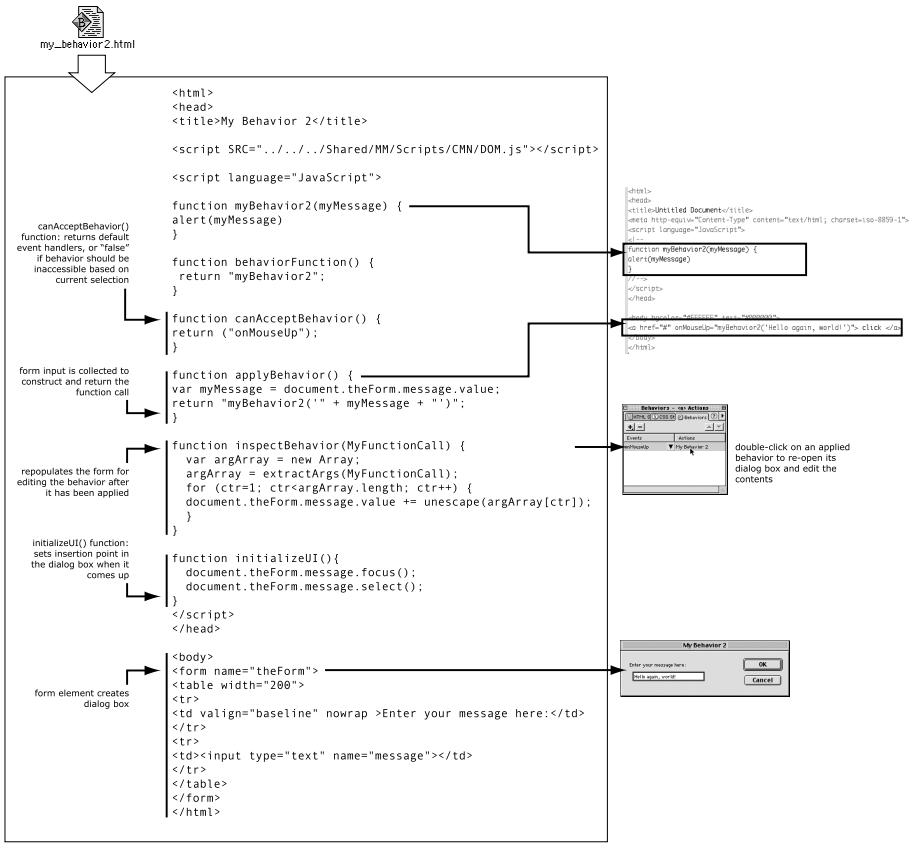


**Figure 22.17** The code framework for a full-featured behavior file.

In addition to the elements listed, the key elements in this more full-featured behavior are as follows:

- **HTML form.** As with object files, the form becomes the dialog box for collecting user input. In behavior files, the form is used to collect arguments for the function call to pass to the main function.
- **Link to script.js file.** This link gives you access to several utility functions for handling strings; one of these functions, the extractArgs() function, is extremely useful in coding the otherwise complex inspectBehavior() function, listed later.
- **canAcceptBehavior() function.** This function, part of the API, returns either true, false, or a list of one or more default event handlers. In more advanced files than the one shown here, this function can also determine what kind of object the user has selected in the document, and whether that object is acceptable for

applying this behavior. If the function returns false, the behavior will appear grayed out in the behaviors list. This function is called automatically.

**Note**

Determining when a behavior should be grayed out in the behaviors list requires more in-depth knowledge of the Dreamweaver API and the Dreamweaver document object model (DOM) than is possible in this chapter. Read the *Extending Dreamweaver* manual and examine other behaviors to learn more about this.

- **inspectBehavior(*functionCall*) function**. Also part of the API, this function must be used to collect information from a behavior that has already been applied and to repopulate the behavior's dialog box for editing. This function makes it possible for the user to open the Behavior panel, double-click an existing behavior, and be presented with a dialog box that contains the current settings for that behavior. The function is called automatically. Dreamweaver automatically passes the behavior's function call, as a string, to the function.
- **initializeUI() function**. This function, which is not part of the API, does the same thing in behavior files as it does in object files: It places the cursor in a certain field of the dialog box and selects any contents for easier text entry. This function must be called using the onLoad event handler in the <body> tag.

In the example shown in Figure 22.17, the code returned by the defined function is a pop-up alert message, but with the content to be determined by the user. Notice that the defined function itself remains generic; the input collected by the form is used to add an argument to the function call.

## Making Your Own Behaviors

In this section, you'll get some hands-on experience making your own behaviors. It should go without saying that you need to be able to write and understand the code for a particular JavaScript function and function call before you can turn that code into a behavior.

### Exercise 22.7 Setting Up Shop

Start by getting your working space in order. As you did with objects previously, you'll create a custom behaviors folder for storing your exercise behaviors, and you'll learn a few of the quirks of reloading extensions when working with behaviors.

1. Make sure that Dreamweaver is running. One of the goals of this exercise is to see how behavior extensions reload.

2. Find and open the Configuration/Behaviors/Actions folder on your hard drive. Minimize or hide Dreamweaver, if necessary, so that you have access to your desktop.
3. Create a custom behaviors folder. In the Actions folder, create a new folder. Call it Development. Leave the folder empty for now.
4. In Dreamweaver, reload extensions. Ctrl/Opt-click the triangle at the top of the Objects panel, and choose Reload Extensions—this is the same procedure you followed when working with objects.
5. In the Behavior panel, access the behaviors list and look for a new submenu. Click the + sign in the inspector to open the behaviors list. A new folder should result in a new submenu appearing in the list.

   Probably, however, you won't see any new Development submenu. Why not? Reloading extensions does not cause Dreamweaver to recognize new or renamed files or folders in the Actions folder; it recognizes only modifications within files. To get Dreamweaver to see the new submenu, you'll have to do it the old-fashioned way: Quit the program and launch again.
6. Quit Dreamweaver and relaunch it. Then look again for the submenu. This time, your submenu should appear. Of course, it will be an empty submenu because the Development folder is empty. But your new behaviors will appear there. Every time you add a new behavior, you must quit and relaunch; when you modify an existing behavior, you can reload extensions without quitting.

### Exercise 22.8 Making a Simple Behavior

A simple behavior is one that doesn't require arguments to be passed from the function call to the function and that, therefore, doesn't need a form to collect user input. The simple behavior that you'll create in this exercise is a script that automatically resizes the browser window to a certain size.

Create and edit the exercise files in your favorite text editor or in Dreamweaver's Code Inspector. Save the behavior file into the Development folder created in the previous exercise. Save the test files in your working folder.

1. Create (and test) the JavaScript function and function call that you want the behavior to insert.

   The first step in creating a successful behavior is writing and testing a stable, functional script for the behavior to insert into a user document. This file is where you will do just that—it isn't going to be the behavior file, so save it in your working folder, not the Development folder. Call it resize400_test.html.

   The document head should include a JavaScript function for resizing the window and also should include a function call from within the body. The code should look like this:

```
<html>
<head>
<title>Testing Resize Script</title>
<script language="JavaScript">
function resizeTo400() {
  window.resizeTo(400,400);
}
</script>
</head>
<body>
<a href="#" onMouseUp=" resizeTo400()"> Click me!</a>
</body>
</html>
```

**Tip**

When writing JavaScript functions to be used as behaviors, you don't have to worry about adding the lines of code that will hide the script from older browsers. Dreamweaver will add those lines of code automatically when your behavior is applied.

2. Test your behavior in a browser (or two). Open the test file in a browser and click on the test link. The window should resize.

   If there's a problem, go back to the code and do some troubleshooting until the window resizes. The script needs to work before the behavior that inserts it will work. For best results, of course, try the behavior in several major browsers before declaring it "well-behaved."

3. Create a basic behavior file, with all structural elements in place. Create a new HTML file. This will be the behavior file, so save it in the Development folder in the Actions folder. Call it Resize400.html.

   Start by entering the framework code, as shown in Figure 22.16. Then add your newly devised function and function call in the appropriate places. Your final code should look like this (elements that have been customized from the basic framework are shown in bold):

```
<html>
<head>
<title>Resize Window to 400</title>
<script language="JavaScript">
function resizeTo400() {
  window.resizeTo(400,400);
}
function behaviorFunction() {
  return "resizeTo400";
}
function applyBehavior() {
  return "resizeTo400()";
}
</script>
</head>
```

```
<body>
<table width="200">
  <tr>
        <td>This behavior will resize the user's browser window to
400 pixels wide and 400 pixels high.</td>
  </tr>
</table>
</body>
</html>
```

4. Relaunch Dreamweaver. If Dreamweaver is currently running, quit the program. Then launch the program again so that the new behavior loads.
5. Create another HTML test file. Call it resize400_behavior_test.html and save it in your working folder. In the new file, create another simple text link (linking to #), just like the first test file. Don't add any JavaScript to this file.

   Your code for the new file should look like this:

```
<html>
<head>
<title>Testing Resize Behavior</title>
</head>
<body>
<a href="#">Click me!</a>
</body>
</html>
```

6. Open the new test file in Dreamweaver and apply your new behavior. When the file is open in Dreamweaver, try to apply your behavior the same way you'd apply any behavior, by selecting the linked text and clicking the + in the Behavior panel. The new behavior should now appear in the behaviors list, under the Development submenu, as shown in Figure 22.18. The dialog box shown in Figure 22.18 should appear when you choose the behavior.

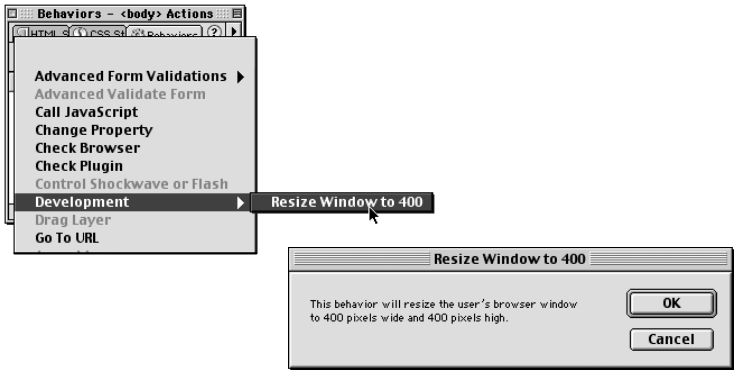


**Figure 22.18** The various interface elements for the Resize Window behavior: the behaviors list, with Development submenu and behavior's menu entry, and the resulting dialog box.

If there's a JavaScript syntax error in your behaviors file, Dreamweaver will give you an error message as soon as you click the + in the panel. Examine the message and see if you can fix the code.

If there are no syntax errors, you'll be allowed to choose and apply your behavior. But this doesn't mean that the code got inserted correctly.

1. Examine the file's HTML source code to see if the script got properly inserted. Open Dreamweaver's Code Inspector and take a look at the code. It should look just the same as the code you entered yourself, back in step 1 in Exercise 22.8.

   If it doesn't, you'll need to do some troubleshooting. Look for the differences between the inserted code and the original, hand-entered code. Then examine your behavior file. Find how the two correlate, and adjust the behavior file. Then try the behavior again.
2. Test the inserted behavior in a browser. In a browser, repeat the test that you performed on the original code back in step 3 in Exercise 22.8. The JavaScript should work just as well as the code you entered by hand. You've made a behavior!

### Exercise 22.9 Adding Arguments to the Function Call

In this exercise, you'll build on the previous behavior. Instead of inserting a script that always resizes the window to the same dimensions, you'll insert a script that asks the user for a desired width and height and then resizes the window to those dimensions. To accomplish this, you'll need to add a form to the behavior file's body and do some fancier scripting in the applyBehavior() function. As in the previous exercise, you'll start by using your test file to create a working version of the function and the function call that you want to insert.

1. Open resize400_test.html and alter the code so that the function call passes arguments to the function. If you want to keep your original test file safe from harm, make a copy of it to work on for this exercise. Save the copy as resize_test.html.

   When you're done, your code should look something like this:

```
<html>
<head>
<title>Testing Resize Script</title>
<script language="JavaScript">
function resizeBrowserWindow(width,height) {
  window.resizeTo(width,height);
}
</script>
</head>
<body>
<a href="#" onMouseUp="resizeBrowserWindow(400,400)">Click me!</a>
</body>
</html>
```

2. Test the revised script in a browser. As before, make sure that the behavior works in as many browsers as possible before declaring it seaworthy. If it doesn't work, troubleshoot until it does.
3. Save the Resize400.html behavior file as Resize.html, and open that file in your text editor.
4. In the behavior file, change the function to match the revised function developed previously. The <script> tag in the head section of the document should now look like this:

```
<script language="JavaScript">
function resizeBrowserWindow(width,height) {
  window.resizeTo(width,height);
}
function behaviorFunction() {
  return "resizeBrowserWindow";
}
function applyBehavior() {
  return "resizeBrowserWindow()";
}
</script>
```

5. Delete the contents of the behavior file's body section, to make way for the form you'll be creating in the next step.
6. In the body section, design and create a form to collect the required information from the user. Your form will need two text fields: one for width and one for height. Code for the new body section of the behavior file might look like this:

```
<form name="theForm">
<table>
  <tr valign="baseline">
        <td align="left" colspan="2" nowrap> Resize the browser
window to these dimensions:</td>
  </tr>
  <tr valign="baseline">
        <td align="right" nowrap>New width:</td>
        <td align="left">
              <input type="text" name="width" size="8">
        </td>
  </tr>
  <tr valign="baseline">
        <td align="right" nowrap>New height:</td>
        <td align="left">
              <input type="text" name="height" size="8">
        </td>
  </tr>
</table>
</form>
```

Remember, you can always design the form in Dreamweaver's visual editor, if you want. The form should end up looking something like the one shown in Figure 22.19.

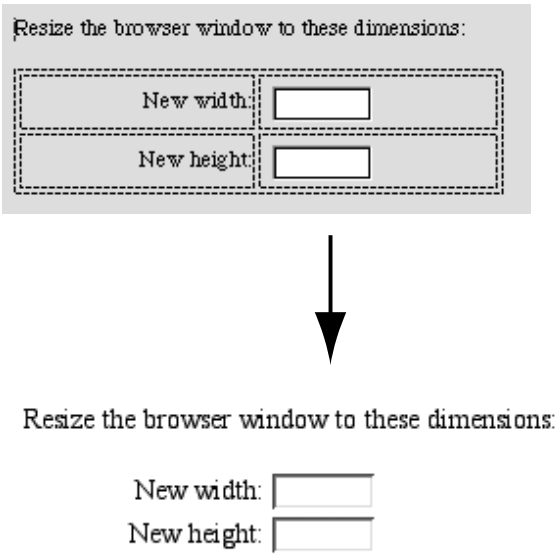


**Figure 22.19** The HTML form for the Resize Window behavior dialog box, as it would appear in Dreamweaver's Layout view and as interpreted by a browser.

5. Rewrite the applyBehavior() function to return a function call that uses the form information. Remember, the applyBehavior() function needs to return the function call exactly as it will appear in the user's document.

   This function's return statement will have to construct the code for your desired function call using concatenation. Your code for the applyBehavior() function should look like this:

```
function applyBehavior() {
   var width=document.theForm.width.value;
   var height=document.theForm.height.value;
   return "resizeBrowserWindow(" + width + "," + height + ")";
}
```

6. Reload extensions and try out your new behavior. In Dreamweaver, create another test file with a text link in it. Reload extensions, and try your new behavior. Does it work? Does it insert the proper code? If not, do some troubleshooting.
7. Refine the dialog box with initializeUI(). Any time there's a dialog box, the local function initializeUI() makes sure that the insertion point lands in a handy place.

   In the behavior file, add the function, specifying the width text field as the focus (it's the first field in the dialog box). It should look like this:

```
function initializeUI() {
  document.theForm.width.focus();
  document.theForm.width.select();
}
```

Also remember to call the function in the <body> tag. The call will look like this:

```
<body onLoad="initializeUI()">
```

### Exercise 22.10 Adding the canAcceptBehavior( ) Function

The most powerful use of this function—determining whether a behavior will appear grayed out in the behaviors list—is beyond the scope of this chapter. But you can specify a default event handler to be used with the function call.

1. Duplicate the behavior file, if you want. If you're afraid of goofing up your lovely working behavior file, make a duplicate of it; remember, though, that you'll have to change the name of the defined function and the page title for the duplicate to work properly.

   If you're feeling brave, go ahead and work on the behavior file that you created in the previous exercise.

2. Add the canAcceptBehavior() function to the head of the file, specifying whatever event handler you want the behavior to use.

   This one's pretty simple. Just add the following code somewhere inside the <script> tags in the document's head section:

   ```
   function canAcceptBehavior() {
   return ("onMouseUp");
   }
   ```

3. Test your revised behavior. You know the drill: Make a test file, reload extensions, and apply the behavior. Examine the code inserted by the revised behavior; the function call should now look like this:

   ```
   <a href="#" onMouseUp="resizeBrowserWindow(300,300)"> click me </a>
   ```

### Exercise 22.11 Creating a Separate JS File

Before you can add the finishing touches to your behavior, you'll need to separate it into an HTML file and a JS file, like you did with Contact Info object.

1. Make a backup copy of Resize.html, just in case. Store the backup outside the Configuration folder, in your working folder.
2. Move the JavaScript functions into an empty text file. Open Resize.html in your text editor. Select everything between the opening and closing script tags, in the head section.

Cut them to the Clipboard. Then create a new, empty text file and paste them into it. The pasted code should look like this:

```
function resizeBrowserWindow(width,height) {
   window.resizeTo(width,height);
}
function behaviorFunction() {
   return "resizeBrowserWindow";
}

function applyBehavior() {
   var width=document.theForm.width.value;
   var height=document.theForm.height.value;
   return "resizeBrowserWindow(" + width + "," + height + ")";
}

function canAcceptBehavior() {
return ("onMouseUp");
}

function initializeUI() {
  document.theForm.width.focus();
  document.theForm.width.select();
}
```

3. Save the new file into the Development folder. Call it Resize.js. Remember, recommended practice is to name the HTML and JS files identically and to store them in the same folder.
4. In Resize.html, add a link to the JS file. Add the link to the <script> tags where you removed the functions. Your <script> tags should now look like this:

   ```
   <script src = "Resize.js"></script>
   ```

5. Reload extensions, and try out your new behavior. As long as all your copying, pasting and linking was correct, the behavior should function the same as it did before. (If it doesn't, check your code.)

   From now on, when you have function changes, you'll change the JS file. The HTML file needs changing only if you update the form, add another function call to the body tag, or add another script tag (you'll be doing that in the next exercise).

### Exercise 22.12 Adding the inspectBehavior() Function

What's the point of this function? Try this: In Dreamweaver, create a test file and apply your behavior. Then take a look at the Behavior panel, where you should see the behavior listed along with its event handler. Double-click the behavior to see and possibly change its parameters.

When the dialog box comes back up, it's empty, as shown in Figure 22.20. Dreamweaver has not retrieved the values that you originally entered into the dialog box, so you have no way of knowing what they are unless you look at the source code.

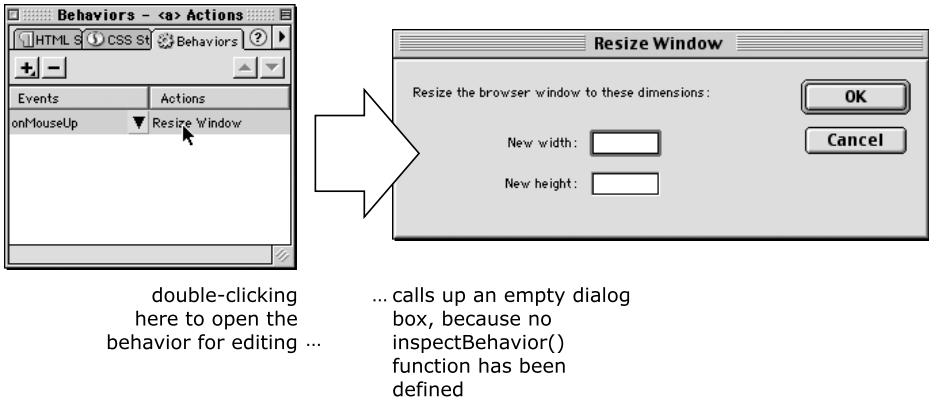


**Figure 22.20** Editing a behavior that has no inspectBehavior() function.

Does this strike you as a real shortcoming? The purpose of this exercise is to fix that problem, by collecting the information from the function call that has been inserted and repopulating the dialog box with that information.

1. Duplicate the behavior files, if you want. Again, if you're afraid of goofing up your existing behavior files, make a backup copy of them, stored in your working folder.
2. In Resize.html, link the behavior file to the string.js file.

   Dreamweaver's Configuration folder contains a bunch of helpful all-purpose JavaScript functions in JS files, ready for you to use in your extension files. You can check them out—they're all in the Configuration/Shared folder. The file that you're going to link to is called string.js—it's in the Shared/MM/Scripts/CMN folder. This file contains several functions for working with strings. It makes the inspectBehavior() function much easier to script.

   To link your behavior file to this JS file, you'll need to add another script tag to the head section of the HTML file, this one containing the relative address of string.js. Your code for the head section of the HTML file should now look like this:

```
<head>
<title>Resize Window</title>
<script src="Resize.js"></script>
<script src="../../../Shared/MM/Scripts/CMN/string.js"></script>
</head>
```

Why did you do this? Because the function you're going to write next (in your JS file) is going to call on several functions from the shared JS file, and these links will allow the two JS files to communicate.

3. In Resize.js, enter the basic framework of the inspectBehavior() function. Start by adding this framework code:

```
function inspectBehavior(resizeFunctionCall) {
}
```

This part of the code collects the function call that the behavior had earlier inserted into the user's document, for inspection. Dreamweaver automatically submits the inserted function call as an argument to the inspectBehavior() function; you just have to enter some sort of argument name, such as resizeFunctionCall (or any other name you like).

4. Extract the arguments from the inserted function call.

Simply by declaring the function, you have collected the inserted function call as a string; now you need to extract the parts of that string that correspond to each argument. This requires some fairly fancy scripting involving substrings—but this is why you linked to the string.js file. That file contains the extractArgs() function, which, when passed a function call, extracts the arguments and returns them as an array. Now all you need to do is pass your collected function call to the extractArgs() function, create an empty array, and feed the return statement from that function into it. The added code looks like this:

```
function inspectBehavior(resizeFunctionCall) {
  var argArray = new Array;
  argArray = extractArgs(resizeFunctionCall);
}
```

5. Use the extracted arguments to repopulate the form (dialog box).

Your new array, argArray, now contains an element for each argument in the function call. In the array, argArray[0] contains the function call itself. The arguments start at argArray[1]; for your behavior, this means that argArray[1] contains the new width of the resized window the behavior is creating, and argArray[2] contains the new height. Because there are only those two arguments in the inserted function call, there are no more elements to the array.

All you need to add now is the code that assigns each array element back into its original form element. For your behavior, the code you need to add looks like this:

```
function inspectBehavior(resizeFunctionCall) {
  var argArray = new Array;
  argArray = extractArgs(resizeFunctionCall);
  document.theForm.width.value = argArray[1];
  document.theForm.height.value = argArray[2];
}
```

That's it! You now have a complete inspectBehavior() function. Your completed code for the behavior should look like that shown in Figures 22.21 and 22.22.

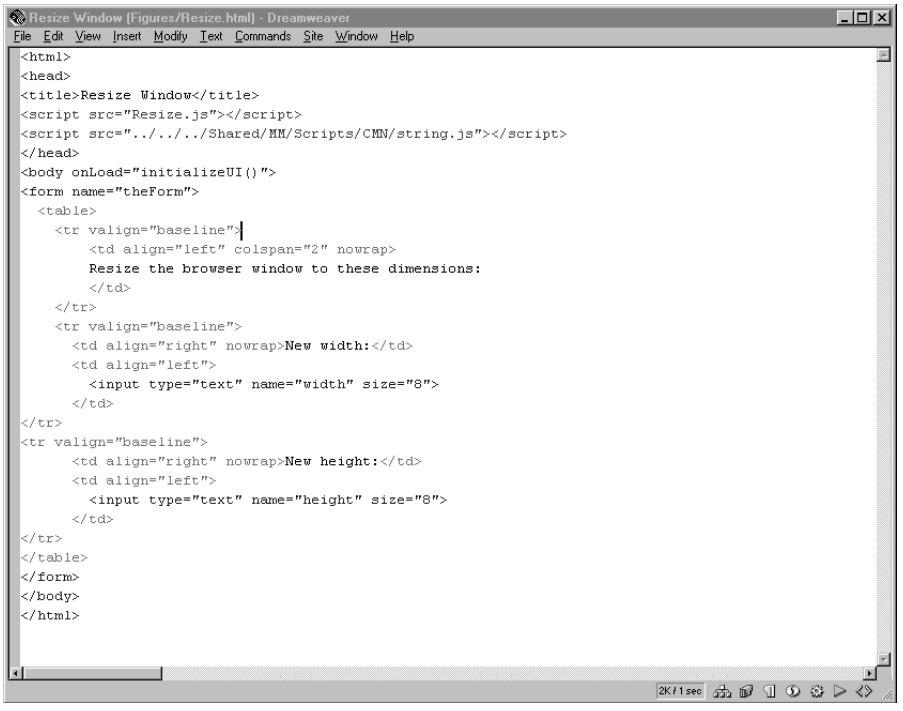


**Figure 22.21** Complete code for Resize.html, with all links in place.

6. Try out the revised behavior. Reload extensions in Dreamweaver and create another test file. Because the code changes are now being made to an external .js file, you can actually use your old test file. Apply the behavior, adding whatever width and height values you like; then try double-clicking its name in the Behavior panel. If everything is working properly, the dialog box should open up with those values in place (see Figure 22.23).

   If you have a syntax error, you'll get an error message.

   If you didn't correctly link to the string.js file, Dreamweaver will bring up an error message saying that extractArgs() has not been defined.

   If you didn't access the array correctly, the dialog box will open with the wrong values in the wrong places, or with empty text fields, or with "Undefined" showing up in one or more text fields.

   If any of these things happen, check and tweak until it works.

File Edit View Insert Modify Text Commands Site Window Help

```
function resizeBrowserWindow(width,height) {
    window.resizeTo(width,height);
}
function behaviorFunction() {
    return "resizeBrowserWindow";
}

function canAcceptBehavior() {
return ("onMouseUp");
}

function applyBehavior() {
    var width=document.theForm.width.value;
    var height=document.theForm.height.value;
    return "resizeBrowserWindow(" + width + "," + height + ")";
}

function inspectBehavior(ResizeFunctionCall) {
  var argArray = new Array;
  argArray = extractArgs(ResizeFunctionCall);
  document.theForm.width.value = argArray[1];
  document.theForm.height.value = argArray[2];
}

function initializeUI() {
  document.theForm.width.focus();
  document.theForm.width.select();
}
```

**Figure 22.22** Complete function code for Resize.js, with the inspectBehavior() function in place.

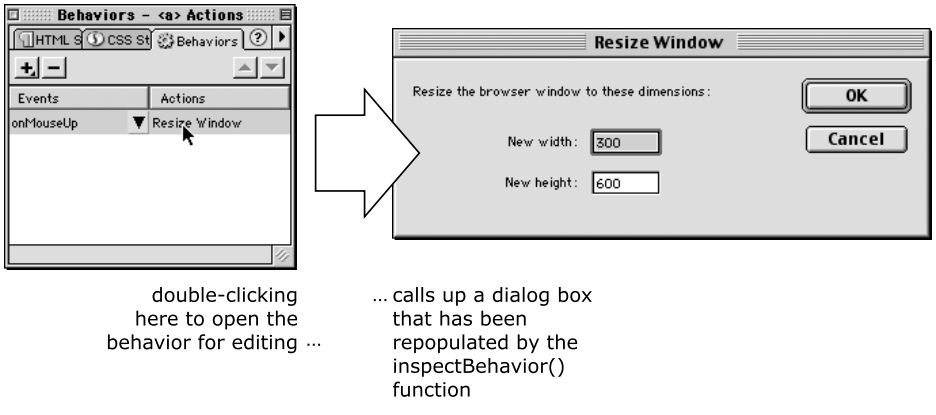


**Figure 22.23** Editing a behavior that has a properly defined inspectBehavior() function.

**Note**

You have seen that extension files should work the same whether the JavaScript code is embedded in the HTML file or is placed in a separate JS file. One exception to this: In behaviors, when linking to shared files (such as string.js), on some platforms the behavior will not work correctly unless the functions are placed in a separate JS file, as you have done here.

### Building Your Own Behaviors

If you've completed all the exercises here, you have the foundation skills to create successful Dreamweaver behaviors. You've also seen that there's a lot more to learn and a lot more that you can do with behaviors. To extend your behavior-making power, check out the *Extending Dreamweaver* manual, examine existing behavior files, and take a closer look at the shared files, such as string.js—these are all valuable resources.

As with objects, though, you should begin by examining your needs. Although it is tempting to create the biggest, baddest behavior on the block—inserting scripts that make browser windows quake and multimedia pour from the computer—it's often the simple, workhorse JavaScript tasks that will give you the most mileage. What JavaScript functionality do you often wish you had quick access to, but don't? Are there existing Dreamweaver behaviors that you wish operated just a little differently? Ask yourself these questions as you work, and you'll soon know what custom behaviors you'll want to add.

## Sharing Extensions

If your object or behavior is helpful to you, maybe it will be helpful to others. Of course, the more people you share with, the more your extension will need to be cleaned up, dressed up, and made reliable and understandable. If you're not sure what this means, take a look at the extensions that ship with Dreamweaver. They're well-documented; the code is bulletproofed; they have standardized, well-designed interfaces; and they're nicely packaged. None of this happens by accident.

### Do You Want to Share?

First of all, you need to ask yourself whether you really want to share. If an object or behavior is probably going to be useful only to your personal workflow, client requirements, or current assignment; if your code is lacking many of the niceties that other people will expect; and if you don't have time to spare getting things in shape to share, you can stop reading right here, and just continue using your custom-made extensions yourself.

On the other hand, you might work in a group setting where all the Dreamweaver users in your office would benefit from your custom extension. Or, you might be a public-minded soul who thinks that the world-at-large can benefit from your brilliance, and you want to submit your extension to the Macromedia Exchange. If that's you, read on!

## Bulletproofing

*Bulletproofing* means making sure that your extension will work under a variety of conditions without breaking. The more diverse circumstances your extension will be used in, the more bulletproof it should be.

How do you bulletproof? Read on.

### *Test the Inserted Code*

No object or behavior is better than the code it inserts. Make sure that your code is worth inserting. Create sample Web pages using code that was inserted with your custom extension. Does the code work successfully across platforms? Does it work in different browsers? Macromedia recommends making sure that your code works in all the major version 4+ browsers and "fails gracefully" in version 3 browsers.

### *Test the Insertion Process*

If there's not a dialog box with opportunities for user input, then the code should insert the same every time. If there is a dialog box, however, then consider the following:

- What happens if the user leaves all values at their defaults and just clicks OK? Does Dreamweaver crash? Does garbage get inserted into the user's document?
- What happens if the user enters unusual or wrong data in the dialog box?
- What do you want to have happen in either of these circumstances? You can code the extension so that valid code gets entered no matter what, you can cause an alert message, or you can simply make the extension not insert code at all unless the user input meets certain criteria. It's up to you.

**Note**

Macromedia recommends that you let the user enter whatever input he desires, as long as it's not going to actually "break" the page—in other words, a result is considered acceptable if the browser will simply ignore the incorrect or nonstandard code rather than generating JavaScript errors, crashing, or exhibiting any other noticeable problems. This is recommended because coding standards evolve—so what's nonsensical code today might be perfectly valid tomorrow—and because users don't like being boxed in by too-rigid requirements for code entry.

A good example of a well-bulletproofed insertion process is Macromedia's own table object. As a learning experience, open Dreamweaver and try using this object with strange dialog box entries. You'll find that the following are true:

- Left to its defaults, the object inserts a valid table based on the last time the object was used.
- Fields that correspond to optional table parameters can be left blank, with the result that the inserted code simply doesn't include those parameters.
- The rows and columns fields, which must have certain values for the table code to function, will not accept invalid entries. Any non-numeric entry, or 0, will be replaced by a 1.

### *Test the Object/Behavior Itself*

You already know that it works on your computer, with your version of Dreamweaver and your operating system. But ask yourself these questions:

- Does it work with older versions of Dreamweaver?
- Does it work on versions of Dreamweaver that are configured differently than yours?
- Does it work on computers that are configured differently than yours?
- Does it work on different platforms?

It may be that some of these things don't matter. If your extension needs to work only in your company, and if you only have PCs running Windows 2000 and Dreamweaver 4, then who cares if it runs correctly on Dreamweaver 2 on a Macintosh or Windows 95? Even though you might not need to fix a certain limitation, though, it's a good idea to be aware of it so that you can share intelligently.

**Tip**

Macromedia requests that all behaviors submitted to the Exchange include, as part of the defined function, the version of Dreamweaver that they are intended to work with. This information should be added as a comment to the defined function, like this:

```
function resizeBrowserWindow(width,height) {  //v3.0
```

### Exercise 22.13 Testing a Custom Behavior

This exercise uses the previously mentioned testing criteria to improve the Resize Window behavior we created earlier in this chapter. (If you don't have the complete code for this object, you can find it on the accompanying CD-ROM.)

1. Test the inserted code. The inserted code, in this case, is a JavaScript function that resizes the current browser window. How robust is this code?

   According to the O'Reilly online JavaScript reference provided with Dreamweaver, this function should work in Netscape 4+ and in Internet Explorer (see Figure 22.24).

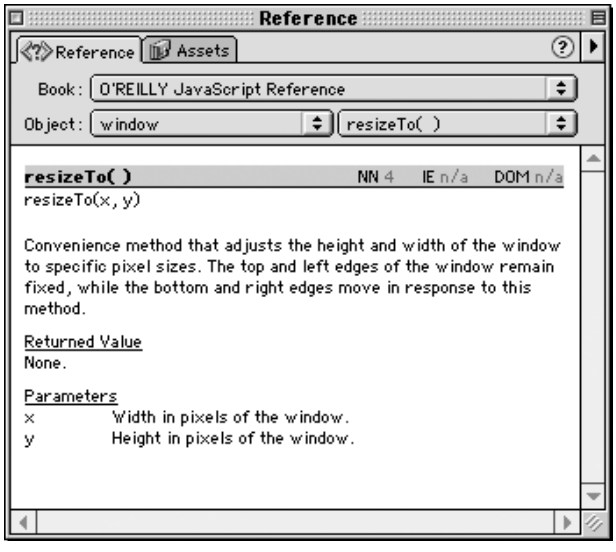


**Figure 22.24** Dreamweaver's Reference window showing the O'Reilly information for the resizeTo() function.

   If you want to be thorough, test this by actually trying the code in as many different browser/platform configurations as you have access to. For purposes of this exercise, assume that the O'Reilly information is correct.

2. Adjust the behavior accordingly. You can't do anything about the script's failure to work in Netscape 3. But you can ask yourself if it will "fail gracefully" in that browser.

   If you try the script in Netscape 3, you'll see that the browser simply ignores it—no harm, no foul. This qualifies as failing gracefully, so there's no adjustment needed.

   See what happens when the behavior is used with default values. As your behavior is currently written, there are no default values for the width and height arguments. What will happen if the user tries to enter the behavior that way? Try it, and you'll see that the code is inserted incompletely:

```
<a href="#" onMouseUp="resizeBrowserWindow(,)"> click </a>
```

The simplest way to avoid this problem is to put default values into the dialog box. Do this by altering the form code so that it looks like this:

```
<table>
  <tr valign="baseline">
        <td align="right" nowrap>New width:</td>
        <td align="left">
              <input    type="text"    name="width"    size="8"
value="300">
        </td>
  </tr>
  <tr valign="baseline">
        <td align="right" nowrap>New height:</td>
        <td align="left">
              <input    type="text"    name="height"    size="8"
value="300">
        </td>
  </tr>
</table>
```

Do this and then try out the revised behavior. The dialog box should always come up with the default values in place.

4. Try the behavior with invalid input. Try the resize behavior and see what it does if you enter non-numeric data, or zeroes, or empty fields, as shown in Figure 22.25. What happens?

   You already know that leaving the fields empty is a dangerous proposition. Entering zeroes is equally dangerous. Entering non-numeric data will generate JavaScript errors in Internet Explorer and is definitely a bad idea. Perhaps you should fix this problem.

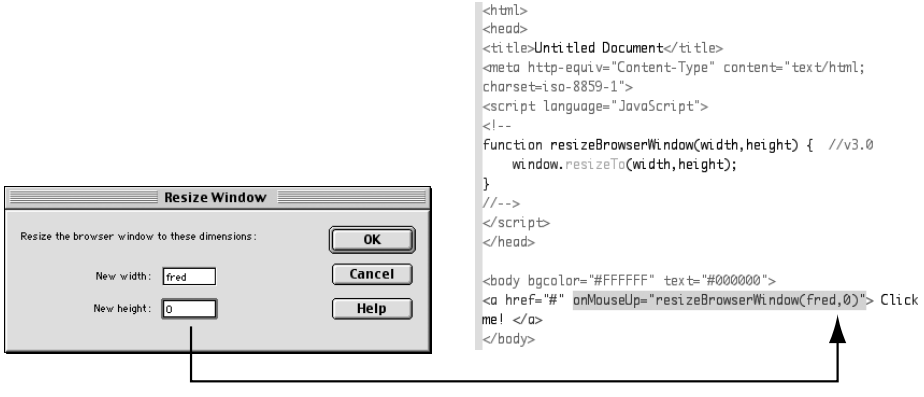


**Figure 22.25** Non-numeric data, or zeroes, entered into the Resize Window behavior dialog box, resulting in invalid arguments passed to the function in the user's document.

5. Rewrite the behavior to disallow invalid input. Rewrite the applyBehavior() function so that the form fields are each validated before the function call is inserted. Valid data should be positive integers only; you also might want to consider extremely large numbers invalid. If no valid data is present, make the behavior resort to its default values.

   Depending on your scripting style, you may choose to implement this error-checking in any number of ways. Your code may end up looking like this:

```
function applyBehavior() {
var width=document.theForm.width.value;
var height=document.theForm.height.value;
if (width=="" || width<1 || width>2000 || parseInt(width) != width)
{
width=300;
}
if (height=="" || height<1 || height>2000 || parseInt(height) !=
height) {
height=300;
}
return "resizeBrowserWindow(" + width + "," + height + ")";
}
```

6. Try out the revised behavior. What happens now when you try to break the behavior by entering some bizarre data (or nothing at all) in the dialog box? It should default to 300×300, or whatever default width and height you decided on. Your behavior is now bulletproof—or, at least, more bulletproof than it was before.

## Design: Testing for Usability

Although you can test for technical errors, the best people to test for design and usability errors are other people—preferably people who have no knowledge of your development process and who are not themselves software developers. Beta testers may indeed find ways to break your extension—in which case, you're back to bulletproofing.

More likely, though, they'll find problems in your design. Is the object or behavior's name self-explanatory, as it appears in menu listings and ToolTips? If it's an object, is its icon communicative and intuitive? If it has a dialog box, is the dialog box attractive and intuitive to use? Does the whole interface blend in well with the Dreamweaver main interface? Is the desired purpose of the extension clear? Does the extension do what users think it's going to do? Is it lacking some key functionality? Do they perceive it as potentially useful?

If the answer to any of these questions is a resounding "No!," you have some redesigning to do.

**Tip**

Macromedia offers a set of UI guidelines to help you create intuitive, functional interfaces that blend in well with the rest of the Dreamweaver interface. To see these guidelines, go to the Macromedia Exchange for Dreamweaver page (`www.macromedia.com/exchange/dreamweaver`) and click the Site Help topic Macromedia Approved Extensions. The code for this chapter has been written to follow these guidelines as much as possible.

## Documenting

So, you think you're going to remember what this extension is supposed to accomplish six months from now or a year from now? Probably not. And if you can't remember it, obviously no one else can, either. Always, always, always document what you're doing—for your benefit and everyone else's.

### *Commenting*

Always comment your code. Always. Macromedia recommends it—you know it's the right thing to do. Commenting will help you troubleshoot and update the object or behavior in the future. It also will help others learn from your process. (The examples shown so far in this chapter haven't been commented so that you could better examine the code for learning purposes. In the real world, they would be full of comment lines.)

### *Online Help*

According to Macromedia, every extension should have some sort of online help, accessible from the extension's dialog box. Dreamweaver is configured to make adding help files easy for you.

- **Help in the dialog box.** If your object or behavior is so simple that a sentence or two is all the explanation that anybody will ever need, you can put that in the dialog box. Macromedia recommends that you add a table cell at the bottom of the layout, with the cell background color set to #D3D3D3. (Figure 22.26 shows an example of this.)
- **Help in a help file.** If your extension needs more explanation than a sentence or two, put the information in an HTML file. Store the HTML file in a new, personalized folder in the Configuration/Shared folder. (Figure 22.27 shows an example of this.) Place a Help button in the extension's dialog box, linked to that file.

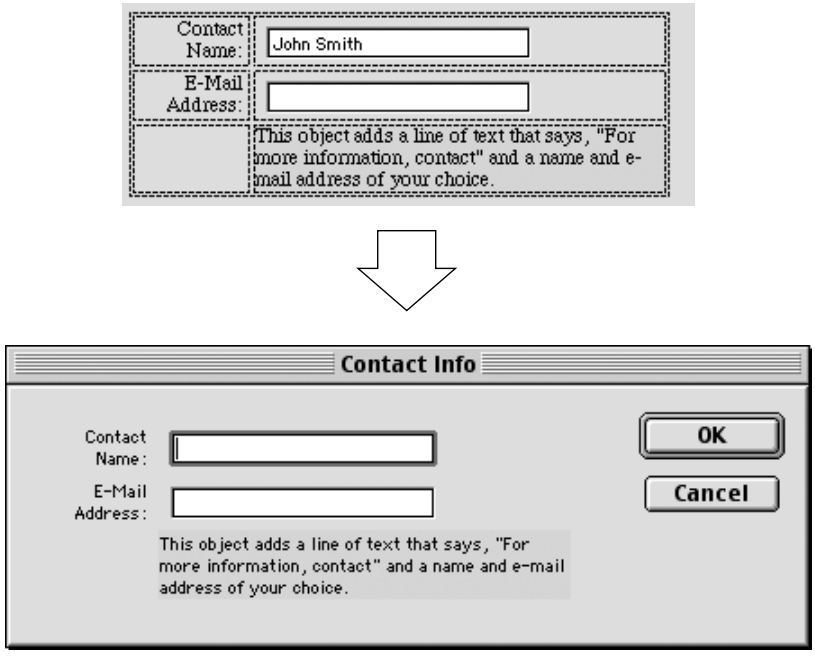


**Figure 22.26** The Contact Info object, with a brief help statement added to the bottom. The top view, taken from Dreamweaver layout view, shows the additional table cell used for formatting.

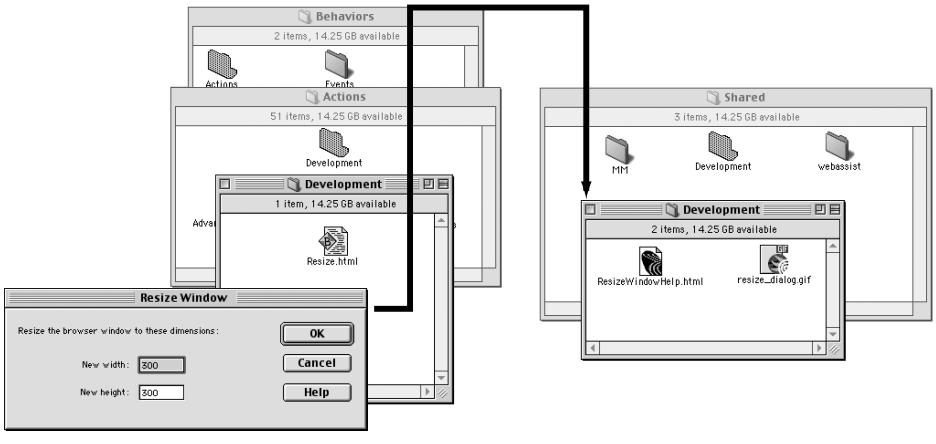


**Figure 22.27** The Resize Window behavior dialog box, with a Help button. Clicking the button tells the behavior file to call on a help file (ResizeWindowHelp.html) in a custom folder (Development) in the Configuration/Shared folder.

## Exercise 22.14 Adding Online Help to Your Behavior

In this exercise, you'll refine your Resize Window behavior by adding a Help button to the dialog box and linking it to an HTML file in the Shared folder.

1. Create a folder in the Shared folder to store your help documents.

   Using Explorer or the Finder, open the Configuration folder and the Shared folder inside it. Create a new folder in here; call it Development. Figure 22.28 shows how the folder structure should look.

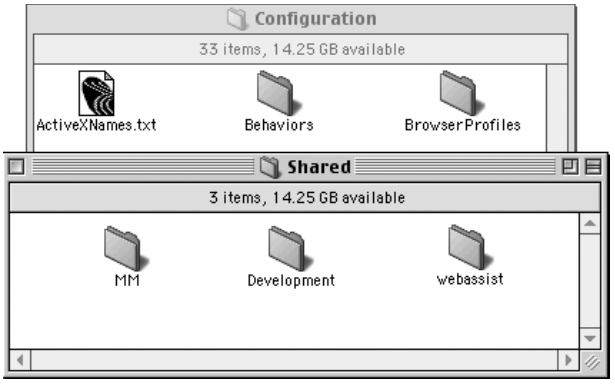


**Figure 22.28** The folder structure of the Configuration/Shared folder with the new Development folder created inside.

2. Create the HTML file. Your help page should include information on what the behavior does, every field and what content it can accept, and any other information you think users will find helpful.

   You can create your own HTML file or use the file ResizeWindowHelp.html, located on the CD. Figure 22.29 shows an example of what a typical help file might look like.

   Save the file as ResizeWindowHelp.html in the Configuration/Shared/Development folder.

3. Add the Help button to your behavior's dialog box. Open the behavior file ResizeWindow.js in your text editor. Add this framework code for the function:

```
function displayHelp() {
}
```

   The displayHelp() function is part of the Dreamweaver API; when present, it is called automatically, so you don't need to call it.

   When you've done this, reload extensions and try out your revised behavior. When the dialog box comes up, there should be a Help button in place, like the one shown in Figure 22.27. (Of course, because the function is empty so far, clicking the button won't get you anywhere.)

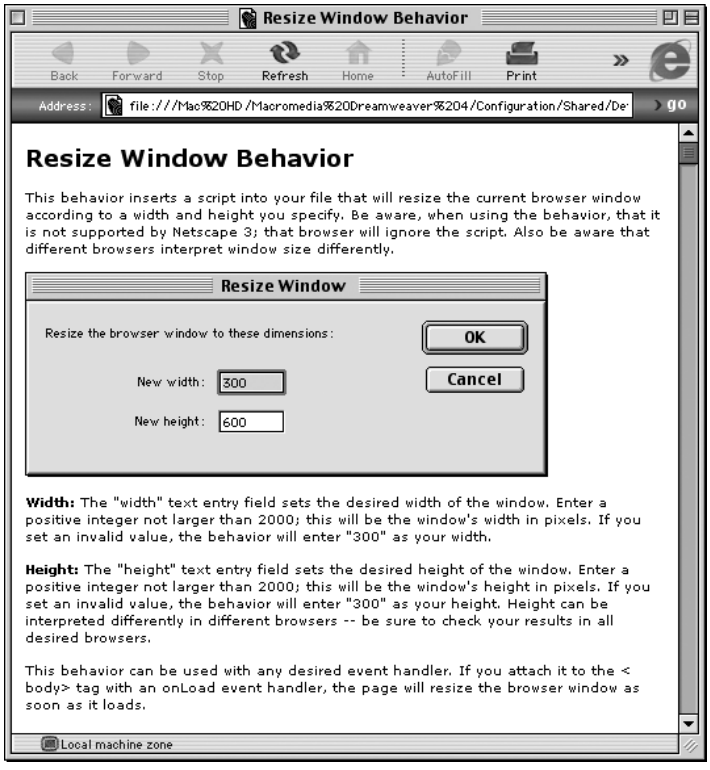


**Figure 22.29** The Resize Window behavior's help file, as it will appear when viewed in a browser.

4. Link the Help button to your help file.

   The standard thing you do with Help buttons is link them to help files. This is done with a function that is part of the Dreamweaver API, dw.browseDocument(). This function takes an absolute URL as its argument. If your help file is located on the Web—maybe on your own company Web site so that users have to come to you to get the latest and greatest help—just enter an absolute Web address as the argument. In this case, the help function would look like this:

   ```
   function displayHelp() {
   dw.browseDocument("http://www.mycompany.com/dwHelpFiles/ResizeWindo
   w.html");
   }
   ```

   Because your help file is going to end up on the user's hard drive, the code needs to return an absolute pathname to that file. Luckily, the Dreamweaver API function, dw.getConfigurationPath(), returns the absolute address to the Configuration folder. All you have to do after getting that information is figure

out the path to the help file relative to this root and concatenate the two together. So, the code you should enter looks like this:

```
function displayHelp() {
var myURL = dw.getConfigurationPath();
myURL += "/Shared/Development/ResizeWindowHelp.html";
  dw.browseDocument(myURL);
}
```

Enter this code. Then, reload extensions and try it out. If the proper help page doesn't load, double-check the code and tweak it until it does. Make sure that you've entered the path from the Configuration folder to your help file exactly—depending on how you've named your files and folders, your path may differ from the one shown here.

**Note**

The two API functions introduced here are both methods of the Dreamweaver object. Methods of this object can be written as dreamweaver.*functionName*() or dw.*functionName*(). The second choice offers fewer opportunities for typos.

## Distributing

How are you going to get your lovely object or behavior into Configuration folders everywhere? Read on for instructions.

### *Packaging for the Extension Manager*

The Extension Manager is becoming the standard method of painless extension installation. Therefore, this is the most accessible way to share your extensions.

Lucky for us, the Extension Manager not only installs extensions, but it also packages them up neatly into special installation files. The process is even relatively painless. The steps are listed here:

1. Put all the required files (help files, HTML files, JS files, and GIF icons) in one folder, outside the Configuration folder.
2. Create an installation file. This is an XML document with the filename extension .mxi that contains all the instructions needed for installation: where the files should be stored, what versions of Dreamweaver and what platforms the extension requires, the author's name, the type of extension, and a description. The formatting required is very exact. The best approach for beginners is to start from the samples included with the Extension Manager. These files include a

blank file (blank.mxi) to use as a template and a sample file (sample.mxi) filled in with information for a simple object.

3. Launch the Extension Manager, and go to File/Package Extension.

See Figure 22.30 for a sample folder containing all the proper files to package the Contact Info file. This last exercise takes you through all the steps to create this folder.

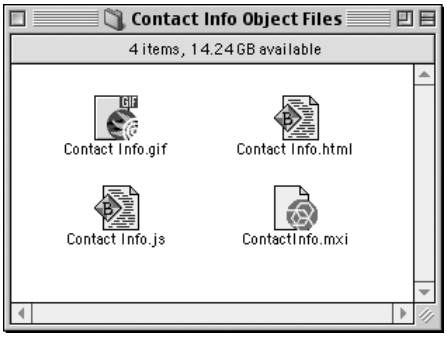


**Figure 22.30** The assembled elements of the Contact Info object, all ready for packaging.

### Exercise 22.15 Packaging an Extension

In this exercise, you'll pack up the Contact Info object for sharing with the world.

1. Copy all needed files into one folder. Somewhere on your hard drive, outside the Configuration folder, create a new folder. Name it whatever you like and will remember (something like Contact Info Files, maybe).

   Find all the files that make up the behavior, and copy them there. Files that you should include are listed here:

   - Contact Info.html
   - Contact Info.js
   - Contact Info.gif

2. Open the blank.mxi file to use in creating the installation file. Duplicate it and save it in your collection folder as ContactInfo.mxi.

   On your hard drive, find the Extension Manager application folder. Inside that folder, find the Dreamweaver/Samples folder. Inside there, you should see blank.mxi. (Figure 22.31 shows where to find these items.)

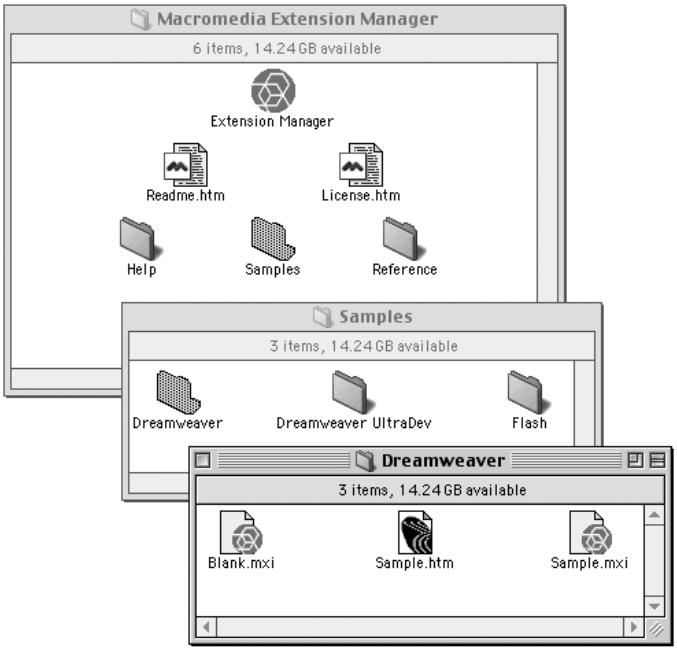


**Figure 22.31** The Extension Manager application folder structure, showing sample.mxi and blank.mxi.

After you've made the duplicate file, open it in your text editor.

**Tip**

You can download a PDF file containing detailed instructions for creating installation files from the Macromedia Web site. Go to the Macromedia Exchange for Dreamweaver page (`www.macromedia.com/exchange/dreamweaver`), and click the Site Help topic Macromedia Approved Extensions.

3. Fill in the blanks with the information for your object. The blank file has all the framework you need. By examining the sample file, you can get an idea how it should be formatted. For your extension, fill in the blanks until your code looks like that shown in Figure 22.32.

   A few tips about filling in the code:

   - **For the author name.** Enter your name (John Smith, Web Genius has been entered here—there's no law against being fanciful).
   - **For the filenames.** Enter the complete path from the Dreamweaver application folder root, as shown. If you want your extension to create any new folders in existing folders, enter them as part of the path (SmithStuff has been entered here to create a new folder within the Objects folder). If the object included any added folders within the Shared folder, they would have been added in the same way.

- **For the version number**. Your extension, like any other piece of software, gets its own version number. Start with 1.0, and increment the number if you later revise the extension.

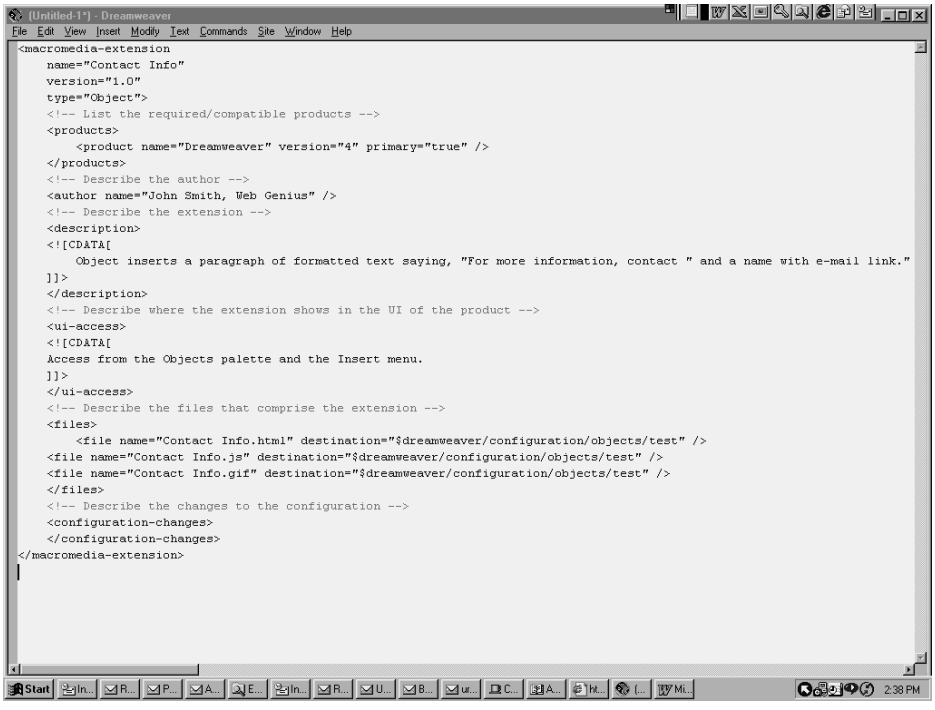


**Figure 22.32** The complete code for ContactInfo.mxi. Information that has been added to the framework from blank.mxi is highlighted.

6. Package everything together with the Extension Manager. Launch the Extension Manager. Go to File/Package Extension.

   For the name of your extension, choose something descriptive that obeys the standard naming conventions (no empty spaces, no more than 20 characters, no special characters). Make sure that you leave the .mxp extension in place.

   When you're asked to choose a file, choose ContactInto.mxi.

   If there aren't any problems, the Manager will generate an extension file in the same folder as the .mxi file. If there are problems, you'll get an error report. Most often, these are problems with the .mxi file. If there are, go back to your text editor, fix the reported errors and try again.

   Figure 22.33 shows how this process will look in the Extension Manager.

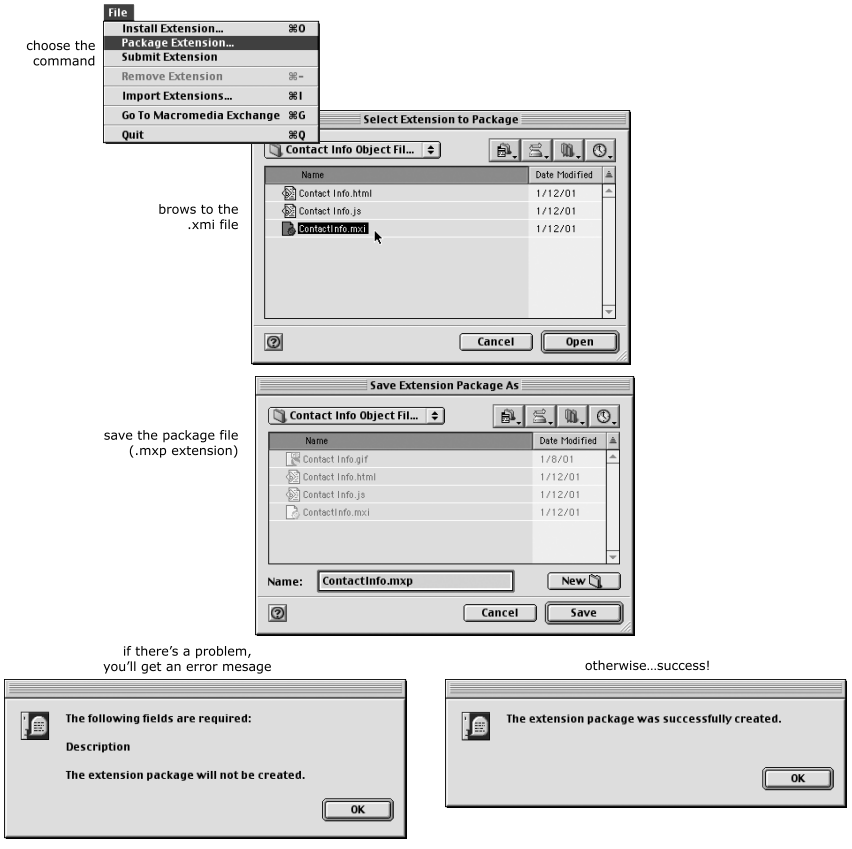


**Figure 22.33** The steps through the packaging process, as they appear in the Extension Manager.

7. Use the Extension Manager to install your new extension. Quit Dreamweaver, if it's running. In the Extension Manager, go to File/Install. When the dialog box comes up, browse to ContactInfo.mxp.

   If everything's hunky dory, you should get an alert message telling you that the extension was installed successfully. Your custom extension should also now appear in the Extension Manager window, as shown in Figure 22.34.

8. Launch Dreamweaver and check that everything installed correctly. If all went as smoothly as reported, a new category should appear in the Objects panel, named SmithStuff or whatever you called your custom folder. Your object should be the only thing in that category. Check the ToolTip; try inserting it. Then pat yourself on the back—you did it!

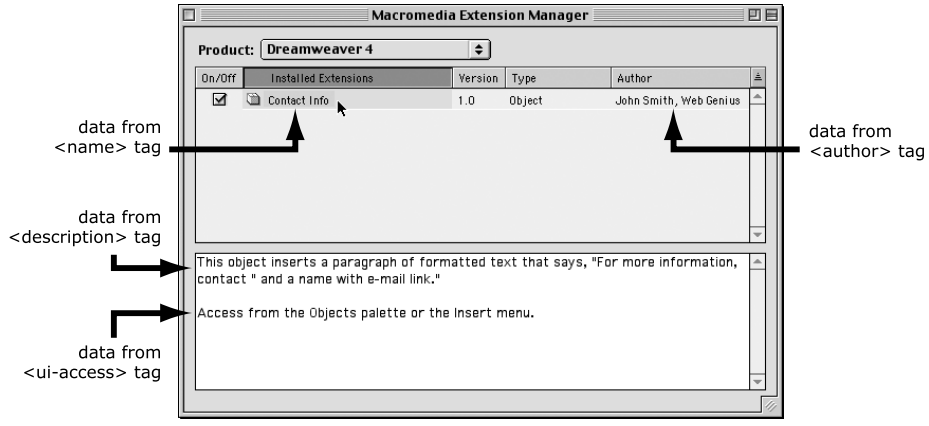


**Figure 22.34** The Extension Manager window, showing the installed Contact Info object.

## *Submitting to the Macromedia Exchange*

The ultimate in sharing is submitting your extension file to the Macromedia Exchange. When you have the .mxp file, the procedure is simple: Go to the Macromedia Exchange Web site and click the Submit button at the top of the page. Then follow the instructions to submit (see Figure 22.35).

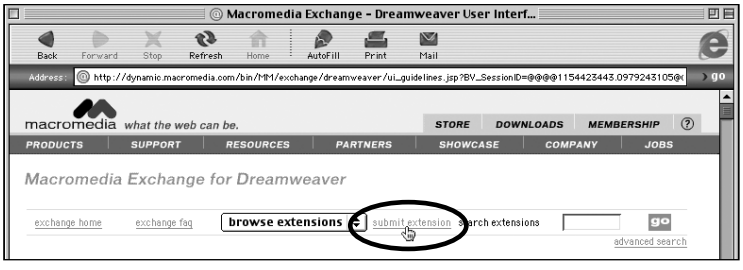


**Figure 22.35** The Macromedia Exchange home page with the Submit button.

When you have submitted an extension, Macromedia engineers will run it through a series of tests. One of three things will happen:

- If it fails, it gets returned to you with comments.
- If it passes the basic tests, it gets put on the Web site as a Basic, or unapproved, extension.
- If it also passes the more comprehensive tests, it becomes a Macromedia Approved Extension.

To learn more about the testing process and how to get your extensions accepted and approved, visit the Web site and click any one of the Site Help FAQ topics. This will take you to an extensive categorized list of questions and answers.

## Summary

You already know that Dreamweaver is a terrific Web editing environment. This chapter has shown you how you can make it into a perfectly personalized Web editor for your workflow needs. As much as you've seen, though, you've only touched the surface of all that is possible with extensions. Check out the *Extending Dreamweaver* manual. Visit the Exchange Web site and read the various support files there. If you're really serious, you can join the Extensibility Newsgroup (go to `www.macromedia.com/support/dreamweaver/extend/form/`). Dust off your JavaScript books. And start rewriting history.