

# JavaScript 1.5

## Table of Contents

INTRODUCTION .....	INTRO-1
A Few Words About This Courseware.....	INTRO-2
World-Class Courseware .....	INTRO-2
What We Expect of You.....	INTRO-2
What You'll Get Out of this Training .....	INTRO-2
Intended Audience.....	INTRO-3
The Practice Files.....	INTRO-4
Installing the Practice Files .....	INTRO-4
What's on the Course CD? .....	INTRO-4
Technical Requirements for the Course .....	INTRO-5
About the Authors.....	INTRO-7
INTRODUCTION .....	1-1
JavaScript's Humble Beginnings .....	1-2
JavaScript and ECMA .....	1-4
New Features of JavaScript 1.5.....	1-5
So Many Browsers .....	1-6
Internet Explorer vs. Netscape Navigator.....	1-8
Introducing the DOM .....	1-9
Recognize Methods, Properties, and Event Handlers .....	1-11
Script Placement .....	1-13
Hiding JavaScript from Older Browsers .....	1-15
External Script Libraries.....	1-16
Viewing Errors in Browsers.....	1-16
Variables, Expressions, and Evaluations.....	1-18
Operators.....	1-19
Simple Conversions.....	1-20
LAB 1: INTRODUCTION.....	1-25
Lab 1 Overview .....	1-26
Dynamic Writing and Event Handlers .....	1-27
Launch External Scripts.....	1-31
JavaScript Links and Calculations .....	1-33
JAVASCRIPT CONDITIONS AND LOOPS.....	2-1

## Table of Contents

---

The if/if...else Control Structure.....	2-2
else...if .....	2-3
The Switch Statement .....	2-5
Boolean Operators: NOT/AND/OR and Bitwise .....	2-8
NOT.....	2-8
AND.....	2-9
OR.....	2-9
Bitwise Booleans .....	2-10
Loops .....	2-11
for Loops.....	2-11
Breaking Out of the Loop .....	2-12
Skipping an Iteration .....	2-12
while Loops.....	2-13
do-while Loops.....	2-14
Using <i>in</i> for Property Looping.....	2-14
The with Statement .....	2-16
Labeled Statements.....	2-16
LAB 2: JAVASCRIPT CONDITIONS AND LOOPS .....	2-21
Lab 2 Overview .....	2-22
Loop the Loop: A Dynamic Table .....	2-23
Decision Structure: Controlling the Table .....	2-29
STRINGS AND FUNCTIONS.....	3-1
Strings.....	3-2
Manipulating Strings .....	3-2
String Concatenation .....	3-2
Changing String Case.....	3-4
Substring Searches .....	3-4
Substring Extraction.....	3-5
Functions .....	3-7
Creating Your Own Functions .....	3-7
Function Parameters .....	3-8
Returning Data from Functions .....	3-9
Variable Scope.....	3-10
LAB 3: STRINGS AND FUNCTIONS .....	3-15
Lab 3 Overview .....	3-16
Build the Page Dynamically .....	3-17
Create a Function for the Calculations.....	3-21

Additional Information .....	3-21
ARRAYS.....	4-1
Introduction to Arrays .....	4-2
Simple Arrays.....	4-5
Arrays as Structures.....	4-7
Parallel Arrays.....	4-12
Multidimensional Arrays .....	4-15
Using the Array Object .....	4-18
Length .....	4-18
Concat .....	4-19
Join.....	4-19
Slice.....	4-21
LAB 4: ARRAYS .....	4-25
Lab 4 Overview .....	4-26
Delimited String to Arrays.....	4-27
Display the Product .....	4-34
FORM INTERACTION .....	5-1
Working with Form Objects .....	5-2
Alternatives to Forms.....	5-2
Form Basics.....	5-4
Form.method Property .....	5-7
Form.action Property .....	5-7
Fieldsets .....	5-8
Form.Elements[] .....	5-10
Text Objects .....	5-11
Button Objects .....	5-15
Check Box Objects .....	5-16
Radio Objects .....	5-17
Select and File Objects.....	5-18
File Input Object.....	5-26
Validation and the onSubmit Event.....	5-28
LAB 5: FORM INTERACTION .....	5-35
Lab 5 Overview .....	5-36
Terminal: Routing and Setup .....	5-37
Defining Validations.....	5-44

## Table of Contents

---

Display Results .....	5-48
BUILT-IN OBJECTS .....	6-1
String Object .....	6-2
Prototype Properties and Methods.....	6-2
charAt().....	6-3
charCodeAt() .....	6-4
fromCharCode ().....	6-5
String.match(RegExp).....	6-5
String.replace(RegExp, string) .....	6-6
String.split("delimiter", [limit int]) or (RegExp).....	6-6
Date() Object.....	6-8
Working with Other Dates .....	6-12
setInterval() and setTimeout().....	6-14
setInterval.....	6-14
setTimeout.....	6-16
Math Objects .....	6-19
The Number Object .....	6-19
The Boolean Object .....	6-21
The Math Object .....	6-21
Math Object Methods and Properties.....	6-22
Regular Expressions and the RegExp Object.....	6-24
Simple Pattern Expressions .....	6-24
The RegExp Object .....	6-28
LAB 6: BUILT-IN OBJECTS.....	6-37
Lab 6 Overview .....	6-38
Terminal: Routing and Setup .....	6-39
Defining Validations.....	6-46
Display Results .....	6-50
WINDOWS AND FRAMES .....	7-1
The Window Object .....	7-2
Spawning a Window .....	7-3
Referencing the New Window.....	7-5
Creating Content in the New Window .....	7-5
Launching Functions and Passing Values.....	7-8
Modal and Modeless Dialog Boxes in Internet Explorer 5+ .....	7-12
Creating a Frameset.....	7-13
Parents and Children .....	7-15

Feb 19 2008 3:29PM Dao Dung dungdq@edt.com.vn

For product evaluation only – not for distribution or commercial use.

**JavaScript 1.5**

Copyright © 2003 by Application Developers Training Company  
All rights reserved. Reproduction is strictly prohibited.



iframes.....	7-19
LAB 7: WINDOWS AND FRAMES.....	7-23
Lab 7 Overview .....	7-24
The Date Selection Frame.....	7-25
Important Date Pop-Up Info.....	7-28
EVENT HANDLING .....	8-1
Popular Browser Event Models .....	8-2
The Sequence of Events.....	8-2
The Event Object .....	8-3
Bubbling vs. Capturing Events .....	8-3
Bubbling Events in Internet Explorer 4+ .....	8-4
Netscape Navigator 4 Event Capture Model .....	8-7
Events in Netscape Navigator 6+ (W3C Compliant Version)....	8-11
Event Objects.....	8-15
The Static Event Object .....	8-15
Standard Event Objects .....	8-15
Browser Differences .....	8-20
Capturing in a Compatible Fashion .....	8-21
Event Types .....	8-24
Mouse Event Types .....	8-24
Keyboard Event Types: text, password, & textarea .....	8-25
Loading/Unloading Event Types .....	8-25
Other Window Event Types .....	8-25
Form-Related Event Types .....	8-26
LAB 8: EVENT HANDLING .....	8-31
Lab 8 Overview .....	8-32
Mouse Interception.....	8-33
Title Cell Info .....	8-38
ERROR HANDLING .....	9-1
Reading Error Messages.....	9-2
Displaying Errors .....	9-2
Investigating Problems.....	9-4
Exception Handling .....	9-6
Try...Catch...Finally Blocks .....	9-6
Custom Error Objects .....	9-9
JavaScript Debugging .....	9-14

## *Table of Contents*

---

The Venkman Debugger.....	9-14
The Venkman Interface.....	9-16
LAB 9: ERROR HANDLING.....	9-27
Lab 9 Overview .....	9-28
Bullet-Proof Functions .....	9-29
Nested Try/Catch .....	9-34
Venkman Test .....	9-37
CUSTOM OBJECTS.....	10-1
Functions .....	10-2
Functions as Objects .....	10-2
Functions on the Fly .....	10-3
Nested Functions.....	10-3
Variables and Arguments .....	10-5
Undefined Parameters .....	10-5
Objects and Properties .....	10-7
Objects and Methods .....	10-10
Arrays, Objects, or Object Arrays?.....	10-11
Prototypes .....	10-14
Method Overriding.....	10-16
Best Practices .....	10-17
Code Refactoring.....	10-17
Template Technique .....	10-20
Creating Organized Libraries .....	10-21
LAB 10: CUSTOM OBJECTS .....	10-27
Lab 10 Overview .....	10-28
Crackers, Photos, and their Methods.....	10-29
Build a User Interface.....	10-34
APPENDIX A: RESOURCES .....	A-1
Books.....	A-2
Web Sites.....	A-3
Tools .....	A-4
Free Editors .....	A-4
Commercial Editors .....	A-5
INDEX.....	INDEX-1

# Introduction

Feb 19 2008 3:29PM Dao Dung dungdq@edt.com.vn

For production evaluation only – not for distribution or commercial use.

**Intro-1**

Copyright © 2003 by Application Developers Training Company  
All rights reserved. Reproduction is strictly prohibited.



# A Few Words About This Courseware

## World-Class Courseware

We have worked very hard to bring you what we think are the best JavaScript training materials in the world!

## What We Expect of You

This course doesn't start at the beginning. To get the most out of this training, you'll need:

- Practical experience with PCs and desktop workstations.
- Knowledge of the most popular browser platforms (Internet Explorer, Mozilla/Netscape, Opera) is helpful.
- General knowledge of HTML/XHTML tags, and especially forms.
- Basic programming knowledge is helpful, but not necessary.
- The desire to learn.

## What You'll Get Out of this Training

Think of this training as a jump-start to creating and working with JavaScript. After taking this class you'll:

- Know how to write JavaScript code and use it in your Web pages.
- Learn how JavaScript interacts with HTML forms.
- Grasp the fundamentals of JavaScript data types, such as Strings, Integers, and Booleans.
- Learn the basics of looping and array handling.
- Learn to handle errors and exceptions gracefully in your applications.
- Discover how you can model data using an object-based approach.
- Organize your JavaScripts to make them easier to change in the future, and implement into other projects.

This course covers the foundational material needed to become proficient using JavaScript with your Web pages. It covers all of JavaScript's critical elements, including language syntax, script design, and deployment. This

course provides a strong foundation in JavaScript, enabling the student to build JavaScript enabled Web pages correctly and with confidence.

## Intended Audience

The training is intended for:

- **Beginning Web designers** who want to know as much about JavaScript as possible to enhance their Web sites.
- **Webmasters** who want to code and debug complex Web projects, and learn to reduce server traffic by manipulating data in the browser with JavaScript.
- **Web developers** who need to expand beyond the limits of HTML.

# The Practice Files

Along with this book, you'll receive a CD-ROM that contains the practice files used in the courseware.

## Installing the Practice Files

Install the practice files by following the instructions on the course CD.

## What's on the Course CD?

The course CD contains:

- **\*.html & \*.js:** These contain the JavaScript examples used in this courseware.
- **Installation files** for the tools you can use during the course, including the Java2 Runtime Edition and jEdit editor, the Venkman JavaScript Debugger for Netscape, Mozilla, and Mozilla FireBird, and the most recent installs for Netscape, Mozilla, Mozilla Firebird, and the Opera browser.

# Technical Requirements for the Course

To write JavaScript code, you'll first and foremost need some form of text editor. You can use Notepad, or any other text editor, but jEdit is recommended due to its advanced syntax highlighting features for JavaScript. You will also need at least one of the most popular browser platforms (Internet Explorer, Netscape/Mozilla, Opera).

Unless specifically noted in the chapter, all the JavaScript examples and exercises will run properly with Internet Explorer, as well as on the various versions of the Netscape, Mozilla, and Opera browsers that have been included on the installation CD for this course.

**WARNING!** You should be aware that there are known incompatibility issues with certain browsers, which may depend on the browser's version number or certain patches from the manufacturers that may have been applied. As a rule, you should test your JavaScript in the specific browser versions that your target audience is likely to use before putting it into production.

Finally, you will need to install the Java2 Runtime Edition, so that you'll have the latest Java plugin on your system.

Here's what you'll need:

- The **Java2 Runtime Edition** (JRE), which is the standard framework for running Java programs. This is needed to run jEdit, the recommended editor for JavaScript. The JRE is available as a free download from the Sun Web site at <http://java.sun.com/j2se/downloads.html>. Identify the latest production release to download, accept Sun's licensing agreement, and select the JRE version for your specific operating system and hardware platform.
- A **text editor** as noted earlier. When you are first learning JavaScript, an editor such as jEdit is ideal. It is written in 100% pure Java and offers features such as JavaScript syntax. The jEdit installation file and instructions are available as a free download from:

<http://www.jedit.org/index.php?page=download>

- A **Web browser** that supports JavaScript 1.5. Internet Explorer 5.5+, Mozilla 1.0+, Netscape 6+, or Opera 7+ are the best choices.

**Mozilla:** <http://www.mozilla.org>

**Opera:** <http://www.opera.com>

**Netscape:** <http://www.netscape.com>

**Internet Explorer:** <http://www.microsoft.com/ie>

# About the Authors

**Neal Ford** is the Chief Technology Officer at The DSW Group, Ltd. He holds a degree in Computer Science from Georgia State University specializing in languages and compilers, and a minor in mathematics specializing in statistical analysis. He is the author of the books *Developing with Delphi: Object-Oriented Techniques*, *JBuilder 3 Unleashed*, and the upcoming *Art of Java Web Development*. His language proficiencies include Java, C#/.NET, Object Pascal, C++, and C, and his primary consulting focus is in building large-scale enterprise applications. Neal teaches classes nationally and internationally to all branches of the military and to many Fortune 500 companies. He is also an internationally acclaimed speaker, having spoken at numerous developers' conferences worldwide. In his spare time, Neal is a voracious reader, an avid music fan, and an Ironman Triathlete. He can be reached at [nford@thedswgroup.com](mailto:nford@thedswgroup.com).

**David Fitzhenry** heads up Graphic Design for The DSW Group, Ltd, including user interface design for Web-based development. Dave's love of all things "Web" began with a teaching assignment on Web development and graphics with Epic Learning. Dave's skill set includes X/HTML, JavaScript, DHTML, and CSS, as well as dynamic Web development with Java and SQL. He is proficient with Adobe's suite of products including Photoshop and Illustrator, Flash MX, Dreamweaver MX, and Director (Shockwave Studio), as well as 3D Studio Max for specialized graphical initiatives. Of late, David has been spending his free time experimenting with Linux and various Open Source projects, and spending any remaining time with his new bride. He can be reached at [dfitzhenry@thedswgroup.com](mailto:dfitzhenry@thedswgroup.com).

**John Grant** is a Software Engineer at The DSW Group, Ltd. He studied Computer Science at the Florida Institute of Technology, with a focus on Software Development. He specializes in the development of multi-tier enterprise applications. His language proficiencies include Object Pascal, C#/.NET, and JavaScript. He can be reached at [jgrant@thedswgroup.com](mailto:jgrant@thedswgroup.com).

**Allan Marks** is a Software Engineer and instructor at The DSW Group, Ltd. He holds a Marketing degree from Temple University, and is a Masters candidate in Computer Information Systems at Georgia State University. His language proficiencies include Java, C#, C++, C, and JavaScript. When he's not busy chasing after his one-year-old twins, Allan specializes in the development of distributed and Web-based enterprise applications using Java. You can contact him at [amarks@thedswgroup.com](mailto:amarks@thedswgroup.com).



# Introduction

## Objectives

- Learn about JavaScript's history.
- Learn how JavaScript and Java are different.
- Understand when JavaScript is the best solution.
- Learn key differences between the popular browsers.
- Learn how to work with simple operators and variables.
- Do simple calculations and conversions with integers.

# JavaScript's Humble Beginnings

Back in the early 1990s, the Web was exploding and browser options were limited. It seemed that every Web site basically contained a lot of text and hyperlinks in the stock Times font with some kind of tiled background image and occasionally littered with animated .gif images.

In 1995, news rang out about a scripting language project from Netscape called LiveScript. The language was developed to serve two purposes. One purpose was to enable Web server administrators to manage Web content from the server side by connecting the pages to resources such as .dll files or databases. The other purpose benefited the client side, allowing Web authors to enhance the user experience by validating forms before submission or communicating directly with Java applets.

Just before the release of Netscape's Navigator 2 Web browser, Netscape and Sun jointly announced that the project was to be named JavaScript. This generated much confusion about the differences between Java and JavaScript.

There are a few similarities between Java and JavaScript. First, they are both interpreted languages. When you download the Java Runtime Environment (JRE), you install the interpreter on each computer that you want Java to run on. JavaScript's interpreter is automatically included in the installation for any modern browser. Second, the syntax of the two languages is similar; both are based on the C and C++ styles. JavaScript supports most of Java's control-flow constructs and expression syntax.

That is where the similarities end. Java supports a full object-oriented programming environment in which you can define your own classes in code and use them to create functioning objects within a program. JavaScript is a scripting language and enables you to write procedural code that can use the objects available to it through the DOM.

In addition, Java is statically typed and strongly type-checked, which means that you must define variables as a particular data type that will be strictly enforced by the interpreter. JavaScript uses a prototype-based object model in which inheritance is dynamic— inherited properties can be different for individual objects.

JavaScript is mainly used for creating more engaging Web pages within the client's browser. In a way, this makes it a cross-platform language, because most hardware/operating system combinations have a browser that supports JavaScript. In Java, you can develop applets that execute within the context of a Web page; and can run in any browsers that have the Java Runtime Environment plug-in installed. You can place objects in XHTML much like an image; they have a height and width and can be inserted nearly anywhere on your Web page.

Support for JavaScript was first introduced in the Version 2 browsers: Netscape Navigator 2 and Internet Explorer 3. At this point, JavaScript was very narrow in scope. It was basically used to perform calculations, provide common programming structures, access form elements, load new pages into frames and windows, and perform date and time calculations.

# JavaScript and ECMA

For years now, Netscape has been working with the European Computer Manufacturers Association (ECMA) to deliver JavaScript as a standardized, international language. The first ECMA standard is labeled ECMA-262 and is currently in its third edition. Standardized JavaScript or ECMAScript behaves identically in all applications that support the ECMA standard. ECMA-262 is also fully approved by the International Organization for Standards (ISO) as ISO-16262.

The ECMA specification defines a common standard for JavaScript and provides a list of requirements to help all developers use the same coding conventions in JavaScript. Table 1 provides a helpful reference if you want to determine whether a particular JavaScript feature is supported under ECMA.

JavaScript	ECMA
JavaScript 1.1	ECMA-262, Edition 1 (based on JavaScript 1.1)
JavaScript 1.2	ECMA-262, Edition 1 (ECMA-262 not complete yet, JavaScript 1.2 was not fully compliant because ECMA and Netscape both added features that were not in ECMA-262 or JavaScript)
JavaScript 1.3	ECMA-262, Edition 1 – Fully compatible (Kept extra features from JS 1.2, except the operators == and != which were added to ECMA-262)
JavaScript 1.4	ECMA-262, Edition 1 – Fully compatible (ECMA Edition 3 was not complete for this JavaScript release)
JavaScript 1.5	ECMA-262, Edition 3 – Fully compatible
JavaScript 2.0	ECMA-262, Edition 4 – Coming soon

Table 1. JavaScript/ECMA comparison chart.

Currently TC39 (the working group of ECMA that is responsible for the scripting standard) is working on the Edition 4 specification, which will correspond to the highly anticipated JavaScript version 2.0.

**NOTE** ECMA-262, Edition 2 consisted of minor editorial changes and a few bug fixes.

# New Features of JavaScript 1.5

From the original JavaScript to the present version, it is safe to assume that JavaScript has come a long way with the explosion of the Web! Many enhancements have made the language more user-friendly, as well as adding and standardizing features that make it simpler to implement.

JavaScript 1.5 includes many technical features as well as a lot of features that simplify code and handle errors. The new features are listed here for your convenience. You do not need to know or even fully understand them at this time.

- **Runtime Errors:** Errors are reported as exceptions, rather than as browser-specific errors.
- **Enhancements to Number Formatting:** New methods are now available for formatting numbers: Number.prototype.toExponential, Number.prototype.toFixed, and Number.prototype.toPrecision.
- **Enhancements to Regular Expressions:** ? can be added after the quantifiers \*, +, ? or {} to make them non-greedy (matching the minimum amount of times). Non-capturing parentheses (?:x) can be used, so that the expression matches x but does not remember the match. The m flag was added to specify that your expression should match over multiple lines.
- **Conditional Function Declarations:** You can now put a function within an if clause. Functions can also be declared within expressions.
- **Multiple Catch Clauses:** You can have multiple catch clauses in a try/catch block to catch any number of exceptions.
- **Getters and Setters:** Java-like getters and setters can be used. This feature is available only for the C implementation of JavaScript.
- **Constants:** JavaScript now supports read-only named constants. This feature is also available only in the C implementation.

# So Many Browsers

With the many available flavors and versions of browsers in the world, it is often extremely difficult to write scripts of even moderate complexity that will work on all browsers. This problem is known as *cross-browser incompatibility* and comes about from differences in the way each browser implements the document object model or DOM (<http://www.w3.org/DOM>). Most of the incompatibilities among browsers are manifested when JavaScript is used to manipulate the DOM in DHTML, which modifies items like images and layers.

While Netscape 6+ or Mozilla 1.2 + are recommended for the exercises at the end of the chapters, scripts in this courseware should work well with any of the version 5 or later browsers that support JavaScript:

- **Internet Explorer 5 +**  
<http://www.microsoft.com/ie>
- **Internet Explorer 5 + for Mac**  
<http://www.microsoft.com/mac/products/ie>
- **Netscape Navigator 6 +**  
<http://www.netscape.com>
- **Opera 7 +**  
<http://www.opera.com>
- **Mozilla 1.0 +**  
<http://www.mozilla.org>
- **Safari (Apple's new Mac browser)**  
<http://www.apple.com/safari>

You will notice that not all browsers are actually in version 5. In Web programming, the term *Version 5 browser* is broadly used to convey the browser's level of support for features like CSS (Cascading Style Sheets) and the DOM (Document Object Model).

Figure 1 shows a Web page rendered in the Opera 7 browser, while Figure 2 shows Apple's Safari Browser in its Beta 2 version. Although these alternative browsers have a slightly different look and feel, they provide similar control layouts and incorporate essentially the same feature set.



Figure 1. Opera 7 browser.



Figure 2. Apple's Safari browser (Beta 2).

In addition, Internet Explorer 5+ for Mac is a separate item in the list because the Macintosh version can be considered a completely different browser from the other Internet Explorer variants. It has a Mac look and feel, and it operates differently with CSS and JavaScript. The Mac version of Internet Explorer is also more compliant with the World Wide Web Consortium's (W3C) standards than the Windows version.

The world of Web browsers is constantly evolving. There are so many browsers that it is truly hard to keep up with the differences. You soon deduce that one of the most difficult things about JavaScript is remembering which features each browser supports.

This course illustrates many of the differences between the two most popular browsers, Internet Explorer (IE) and Netscape Navigator (NN). You'll find that it is much easier to write JavaScript code that the other browsers support once you learn to write compatible scripts for these two browsers.

## **Internet Explorer vs. Netscape Navigator**

Internet Explorer and Netscape Navigator were the two principal adversaries in the original “browser wars,” and each constantly competed to top the other with new features. With Netscape’s introduction of JavaScript, Microsoft saw that its rival had made a very powerful contribution to the Internet world. Microsoft would have preferred that VBScript become the browser-scripting standard, since it is conveniently provided in the Windows versions of IE. However, the scripters of the world tended to migrate toward JavaScript because more browsers support it across a wider range of computing platforms.

With IE 3.0, Microsoft launched its own variation of JavaScript, called JScript. JScript was a superset of JavaScript, including virtually all of the same functionality, but with additional features that could take advantage of enhanced functionality that Microsoft built into Internet Explorer.

Today it seems that the war has cooled a bit, but both camps are still boasting about various features like “We are more standards compliant!” and “We integrate better with the operating system!”

When trying to decide which JavaScript features to use in your Web pages, the questions you will want to ask yourself are:

- Is that cool feature going to work across different versions of the most common browsers?
- Do I want to maintain multiple versions of my Web site to support browser-specific features?
- Will this feature impede navigation for someone whose browser doesn't support JavaScript, or who has JavaScript support turned off?

These are all very important questions that you'll need to answer. Until all the browsers standardize on W3C's DOM specification, the safest way to reach the most browsers is to keep your scripts simple. When you script for form objects and other objects in the DOM, be sure to test your code for cross-browser compatibility unless you are absolutely certain that it will work with all the browsers you are targeting.

# Introducing the DOM

When you view a Web page, everything you see in the window of your browser is based on the DOM, or document object model. The DOM contains every object on the page. Common objects are form objects like buttons, text areas, select objects, and radio buttons. Objects can also be outside of forms, as are images or layers.

Figure 3 shows the basic DOM. To use JavaScript properly, it is important to understand the hierarchy of the DOM.

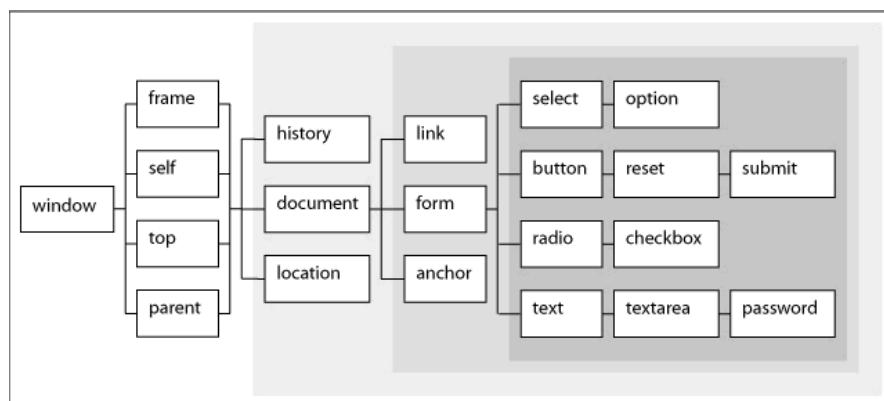


Figure 3. Basic DOM for all JavaScript enabled browsers.

At the far left side of the figure is the top object (window) in the DOM hierarchy, and when it has focus your scripts can operate. To the right of window you'll find frame, self, top, and parent, which are essentially page references. Assuming that the active window is not a spawned pop-up window, it would also be the parent window, which can spawn new child windows or pop-up windows. In a frameset, the original window that sets up the individual frames is the parent.

In JavaScript you refer to these objects by using *dot syntax*. The roots of dot syntax go all the way back to C. Basically, it means that a period separates objects in a hierarchy. For example a line of text, or string, that you write to the browser in JavaScript would look like the following:

```
<script language="JavaScript" type="text/javascript">
    // Complete "formal" version:
    window.document.write("I'm text in the browser!");
    /* Informal "shorthand" version - works only if the
       parent window has focus: */
    document.write("I'm text in the browser");
</script>
```

You reference it almost like the chart in Figure 4.

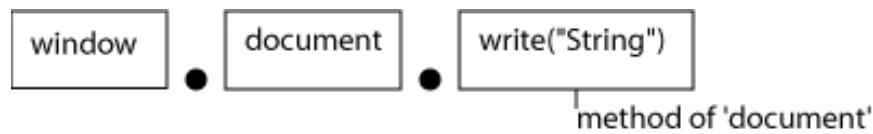


Figure 4. A dot syntax example.

Dot syntax is used everywhere on the Web. For example, in the URL [www.appdev.com](http://www.appdev.com), www is the prefix, appdev is the domain address, and com is the extension. You can actually use the DOM and JavaScript to get the URL or href of the current page. Try typing this in the address bar of your browser:

```
javascript: window.alert(window.location.href);
```

Notice that window.location.href returns the current URL in the address bar of the browser window.

For example, you could reference a form in the DOM through the document object by using the following syntax:

```
document. [formname]
```

Similarly, you could reference an object within the form with the following syntax:

```
document. [formname] . [objectname]
```

Once you have a reference to an object, you can use it to work with that object and access its properties or call its methods.

## Recognize Methods, Properties, and Event Handlers

Almost all objects in XHTML and even JavaScript have properties and methods, with the occasional exception of objects that have only one or the other, but not both. Furthermore, objects might have event handlers, which detect a particular event within the browser, like a button click or a mouse movement.

A property is basically an attribute or setting of an object. In the previous alert example, when the browser was loaded the location object was passed the actual URL string as the href property. When window.location.href executed, it passed the URL string to window's alert method, which then displayed it in a message box. In an earlier example you saw document.write("string"), which wrote the string's value on the page.

**TIP:** You can also write XHTML within the boundaries of a string in JavaScript.  
For example: document.write("<b>Hello World</b>");

The .write part of this statement is actually a reference to the write method of the document object. When called, the write method expects a variable to be passed to it within the parentheses. With JavaScript's loose typing, this variable could be a string, an integer, a boolean, or almost any JavaScript object (as long as it supports construction). Another example to type in your browser's address bar is:

```
javascript: window.open("http://www.yahoo.com");
```

This fires the window object's open method or, as some people like to call it, the "annoying pop-up window" method. Many ISPs are advertising a service they've developed to block the results of this little JavaScript method!

Without venturing too far into event handlers at this point, it will help you identify standard XHTML event handlers. Typically, you can attach event handlers to various form objects. A simple text input object looks like this:

```
<input type="text" name="textFieldObjectName"  
      value="default value"/>
```

Now, suppose that in your JavaScript you want to show a message when the value property of this text input has changed. You could use the input object's onChange event to accomplish this goal:

```
<input type="text" name="textFieldObjectName"  
value="default value" onChange="window.alert('I  
changed!')"/>
```

Here the onChange event is defined as a property of the input object, which fires when the value in the text field changes. Once this object loses focus on the form (when the user clicks in or on another object), the text field will register any value change and fire its onChange event, which in turn will fire the window's alert method.

Another popular XHTML/DOM event is a form object's onSubmit event. This is a great tool for validation because it fires before the form is actually submitted to the action address.

This is a very basic introduction to the DOM and the many objects it contains. You will learn more in a later chapter about the importance of mastering DOM.

# Script Placement

You cannot write scripts without knowing where they go! Proper script placement is a balance between timing and organization. The browser reads XHTML line-by-line from top to bottom; this means that your JavaScript is picked up in the same order that you place it in the script. In addition, a script can be embedded within an XHTML document, or placed in a completely different file and then referenced by the XHTML document. Here is the beginning of a simple XHTML document:

**NOTE** If you do not know XHTML, don't worry. It is nearly the same as HTML, except that you must make sure each opening tag has a matching closing tag. Even single tags like <br> must be closed with <br/>, which is known as a self-closing tag. This makes the document a valid XML style document.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
...."-//W3C//DTD XHTML 1.0 Strict//EN"
....."http://www.w3.org/TR/xhtml1/DTD/xhtml1
.....strict.dtd">
<html xmlns=http://www.w3.org/1999/xhtml xml:lang="en"
lang="en">
<head>
....<script language="JavaScript" type="text/javascript">
.....var googleurl = "http://www.google.com";
....</script>
</head>
```

In this example, the <?xml ...> tag tells any browser or XML parser that this is an xml 1.0 document, and the <!DOCTYPE> tag gives parsers a DTD to validate the document against.

Note that this document also defines a JavaScript *global* variable, googleurl, in the <head> of the XHTML document. If a script is defined above the global variable in the document, that script cannot access the variable unless it defines a function that can be used after the complete document is loaded.

It is best not to use global variables in a complicated script because it can become difficult to track the value of such variables while many functions are accessing them.

**WARNING!** A global variable is a variable that *any script* on the page can use. If you use global variables, be extremely careful not to modify the value of a variable in one function that other functions rely on for their calculations. This can lead to bugs in your code that can be very difficult to track down and fix.

Continuing in this document, you may want to create a link with the `googleurl` variable which was defined in the `<head>` of the document:

```
<body>
....<script language="JavaScript" type="text/javascript">
.....document.write("<a href=\"\"");
....document.write(googleurl);
....document.write("\">Proceed to Google!</a>");
....document.close();
....</script>
</body>
</html>
```

**NOTE** To allow a double quote within a string without breaking the string, type a backslash (also known as the escape character) plus the double quote: \"

The `document.write` statements in this example would translate to the following in the XHTML document when the script executes:

```
<a href="http://www.google.com">Proceed to Google!</a>
```

In addition, you'll notice that the script calls the `document.close` method. This is a good idea when you are using `document.write` several times in a script. The `document.close` method explicitly closes the writing stream that the JavaScript opened to transfer text to the XHTML page.

Some people like to keep their scripts tidy and hidden away in the head section. However, you may want to display a message once the page has loaded without putting `<script>` tags at the bottom of the page. The best way to accomplish this is with an event handler. Like many form elements, the `<body>` tag of the document has event handlers. In this case you'll want to use the body tag's `onload` event. The body also has an `onunload` event handler for when the page unloads and is ready to move to the next link or address.

```
<body onload="alert('Hi! This page has loaded.')">
// and
<body onunload="alert('Y'all come back now, hear?')">
```

With JavaScript it is important to look into all the options before trying to implement a convoluted script to solve a problem. JavaScript has been around for some time now and has matured gracefully, so chances are there is a simple solution to whatever problem you are trying to tackle.

## Hiding JavaScript from Older Browsers

Because browsers earlier than Version 3 do not support JavaScript, they are incapable of handling it properly. If a pre-Version 3 browser accesses a page with JavaScript, the script itself displays on the page. To avoid this potential problem, you will want to write the following lines into your scripts:

```
<script language="JavaScript" type="text/javascript">
<!--
....document.write("Actual script here!");
//-->
</script>
```

The preceding code uses XHTML comment tags to wrap the JavaScript code, which effectively hides it from an older browser. The opening XHTML comment tag, <!--, appears on the line above the JavaScript code that you want to hide, and will be treated as a one-line comment. On the line following the JavaScript code, you have to use the single-line comment // and end the original XHTML comment with -->.

Modern browsers will interpret the opening tag and closing tag as two separate comments, and will interpret the JavaScript code properly. However, pre-Version 3 browsers will see these as one tag, and will simply ignore the JavaScript code contained between them. In addition, pre-Version 3 browsers will also ignore the surrounding <script></script> tags, so they won't pose a problem.

## External Script Libraries

In addition to embedding JavaScript in an HTML document, you can also place your scripts in external files called libraries. A library is basically a text file with a different extension (.js). The placement of a library file reference works the same way as other file references in HTML: you use a relative path to the file, so the browser knows where to look for it. If no path is specified, the browser will look in the same directory as the HTML document that references the script file.

Code to insert a JavaScript file, MyScript.js, in the same directory as the .html file would look like this:

```
<script language="JavaScript" type="text/html"  
src="MyScript.js"></script>
```

A typical library file might look like this:

```
<!-- block script from old browsers  
var w3curl = "http://www.w3c.org";  
window.location = w3curl;  
// -->
```

Why would you want to use separate JavaScript files? The simple answer is: organization. Upon completing this course, you may write some JavaScript objects and methods that will be useful later on in other projects. Saving these methods and objects in library files that you simply copy over and attach to another .html project is a very efficient way of scripting. You will save yourself time and make it easier for other developers to reuse your code or to maintain that site. In addition, the majority of the scripters in the world share their work in the public domain to save JavaScript newcomers some time on the learning curve.

## Viewing Errors in Browsers

While you try out all this scripting, you may want to know how your browser reports problems. All modern browsers use error windows to report any problems they encounter when rendering a page, as shown in Figure 5.

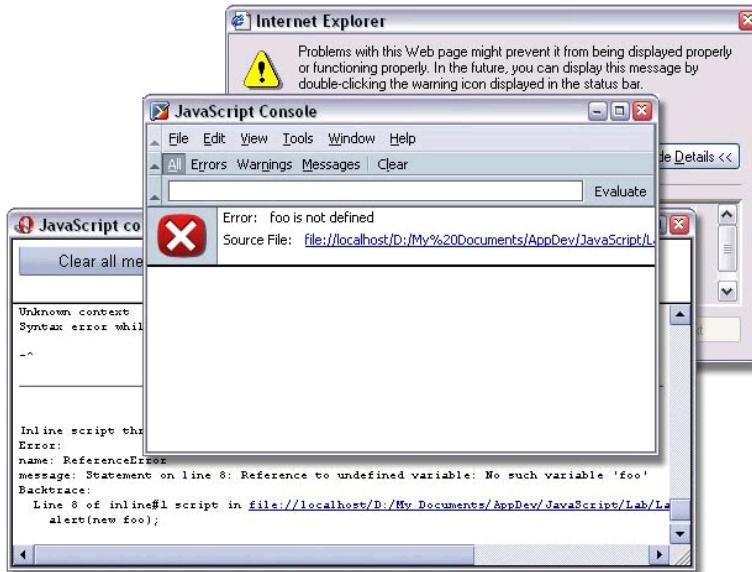


Figure 5. Error windows.

In the beginning you might see error windows like these quite often. Don't worry though; it gets easier. You will find that Mozilla has a very helpful error window. It does not pop up immediately when an error occurs, but once your page has loaded you can choose **Tools|Web Development|JavaScript Console** to view it. Once the window is open, you can enter other JavaScript statements in it and evaluate them. In addition, once you become a little more adept, you can install the Venkman JavaScript debugger in Mozilla, which is available at the following URL:

<http://www.mozilla.org/projects/venkman/>

This handy tool will allow you to place breakpoints in your code. Once the code runs, the project will stop execution when it encounters the breakpoint and you can investigate the values of the different variables at that point in time. You can also step through your code, to see how those variables change under different conditions or with different input. This is handy for complicated scripts, especially when you encounter global variables like those mentioned earlier in the chapter.

# Variables, Expressions, and Evaluations

Data is what JavaScript is all about. Every time you write some JavaScript, you are working with data. That data, of course, comes in many forms. As shown in Table 2, JavaScript defines five and a half main data types; the extra half refers to the fact that a function serves quite well as an object, as you will see a bit later!

Name	Example	Description
Boolean	true	Logical true or false
Null	null	No value, whatsoever
Number	29.2	A number not in quotes
Object	[object, object]	An abstract data type
String	“foo!”	Series of characters in quotes
Function	function(){} function foo() {}	Callable statement/object-like

Table 2. JavaScript data types.

Expressions are statements that evaluate to a value. To see how JavaScript could be used to evaluate the values of variables, consider the following example:

```
<script language="JavaScript" type="text/javascript">
....var age = 29;
....var lastNumBefore30 = (30 - 1);
....var isOld = false;
....if (age >= lastNumBefore30) {
.....isOld = true;
....}
</script>
```

In the code above, the variable `age` is assigned the value of 29, and the variable `lastNumBefore30` is assigned the value of  $(30 - 1)$ , which evaluates to 29. Therefore, the expression `(age >= lastNumBefore30)` evaluates to true, so the statement within the if code block will be executed, and the value of the variable `isOld` will be set to true.

Though it is not fair to pick on 30-year-olds, the example is fairly clear. Each line in this example is an expression, and at the end of the script each statement has been evaluated.

## Operators

Operators are used to help evaluate statements in whatever way you require. The previous example made use of arithmetic operators and comparison operators. As you might guess, arithmetic operators perform basic mathematical functions, such as:

```
10 - 10 + 10 / 10 * 10 = 10
```

Arithmetic operators can also work with strings, albeit a little differently:

```
USA = "United" + " " + "States" + " of " + "America";  
// result = "United States of America"
```

In programming terminology, this is called string concatenation, where each subsequent string is appended to the string before it.

Comparison operators are used to make decisions within a program. As you saw in an earlier example, the `>=` operator was used in an if statement to determine whether the variable `age` was greater than or equal to the variable `lastNumBefore30`.

The JavaScript comparison operators are:

- `>` Greater than
- `>=` Greater than or equal to
- `<` Less than
- `<=` Less than or equal to
- `==` Equal
- `!=` Does not equal

Note that when you want to test for equality in JavaScript, you must use the double equal sign:

```
if (30 == 30) {  
....alert("true");  
}
```

A single equal sign performs an assignment rather than an evaluation, such as assigning the value 29 to the variable lastNumBefore30:

```
lastNumBefore30 = 29;
```

## Simple Conversions

In JavaScript, you will need to learn how to convert various data types into other data types, and back again. As you will see, this is a process that you will find useful, particularly when working with forms. First, take a look at the following example, which shows how to convert a number into a string:

```
fossilAge = 40 + "";  
//result = "40"
```

In this case, 40 is an integer value. However, the code uses the + operator to concatenate the integer value and the empty string (""). Adding a non-string variable to a string always results in a string.

Note that a conversion may be complex and require many levels or steps. To help ensure that you get the results you are looking for, you can break the conversion down by using parentheses. In the following two statements, note how parentheses can be used to group the same input values to get very different results:

```
conversionTest1 = 4 + "0" + 12 + 123;  
// conversionTest1 equals "4012123"  
  
conversionTest2 = (4 + "0") + (12 + 123)  
// conversionTest2 equals "40135"
```

The results differ because in conversionTest1, the values are handled in sequence from left to right; each character is converted to a string and appended to the end of the preceding result. In conversionTest2, the

expressions within the parentheses are evaluated first. If the parentheses contain numeric values, the calculation is performed first, followed by the string conversion and concatenation.

You will definitely need to know how to convert strings into numbers. Any time a user enters numeric values in a text field in a form, these values will be held as strings. To use these values in a computation, you must first convert the string values to their numeric equivalents. Fortunately, JavaScript has helpful methods to assist you with this task.

For a regular int:

```
....var stringInt = "1200";
....var anInteger = parseInt(stringInt);
....// value of anInteger: 40
```

For a floating decimal point (or any decimal point):

```
....var stringFloat = "1200.1242";
....var aFloat = parseFloat(stringFloat);
....// value of aFloat: 1200.1242
```

By now you may be wondering, “If the user can enter whatever value they like in a text field, how can I be sure I’ll be dealing with a number?” When using parseInt and parseFloat you may first want to make sure the incoming string is truly a number, or you could get undesirable results. You can verify that the string is a number by using the method isNaN, which stands for “is Not a Number”. The isNaN method returns a Boolean value of true if the value cannot be converted to a number, or false if the value can be converted to a numeric equivalent:

```
....var stringFloat = "1200.124whoops!";
....var aFloat = 0;
....if (isNaN(stringFloat) == false) {
.....aFloat = parseFloat(stringFloat);
....}
```

In this case stringInt.isNaN returns a Boolean true because the characters whoops! are located within the string. By first checking whether stringFloat can be converted to a number, you can avoid the error that would occur if aFloat is not convertible.

# Summary

- JavaScript evolves constantly with the evolution of the Web.
- JScript and JavaScript both support the core ECMA standard.
- There are many JavaScript capable browsers, each with their own features and flaws.
- You must understand the DOM and dot notation, which represents the layout of the objects in your XHTML document that you can access with JavaScript.
- Script placement is a blend of organization and timing. The XHTML page and JavaScript are read from the top down.
- Variables in JavaScript are loosely typed.
- Strings are added to or concatenated with regular operators.
- Use isNaN to test whether a string contains characters that cannot be converted to a numeric equivalent.
- Convert numbers in a string to actual numbers by using the parseFloat and parseInt methods.

# Questions

1. Which two companies decided on the name JavaScript?
2. Which methods can you use to convert a string into a number?
3. Which method can you use to verify that a string has only numbers?
4. External JavaScript files should have what extension?
5. A variable defined outside of a function in the context of the XHTML page is considered to be a \_\_\_\_\_?
6. What method of the body tag fires only after the page is loaded?
7. What does the DOM acronym stand for?
8. What kind of syntax does JavaScript use to call the methods of an object?
9. What is the name of Microsoft's implementation of JavaScript?
10. Who defines the standardized JavaScript specification?
11. How have errors changed in JavaScript 1.5?

# Answers

1. Which two companies decided on the name JavaScript?  
**Netscape and Sun Microsystems**
2. Which methods can you use to convert a string into a number?  
**parseInteger and parseFloat**
3. Which method can you use to verify that a string has only numbers?  
**isNaN**
4. External JavaScript files should have what extension?  
**.js**
5. A variable defined outside of a function in the context of the XHTML page is considered to be a \_\_\_\_\_?  
**global**
6. What method of the body tag fires only after the page is loaded?  
**onLoad**
7. What does the DOM acronym stand for?  
**Document Object Model**
8. What kind of syntax does JavaScript use to call the methods of an object?  
**dot syntax**
9. What is the name of Microsoft's implementation of JavaScript?  
**JScript**
10. Who defines the standardized JavaScript specification?  
**ECMA**
11. How have errors changed in JavaScript 1.5?  
**Errors now throw exceptions, instead of just being browser based.**

# Lab 1: Introduction

**TIP:** Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You'll find all the code in **HelloWorld.html**, in the Completed subdirectory for this lab. To avoid typing the code, you can cut/paste it from the source file instead.

# Lab 1 Overview

In this lab you'll learn where to properly place your scripts, how to include an external library script, and how to work with simple variable conversions and arithmetic.

To complete this lab, you will need to work through three exercises:

- Dynamic Writing and Event Handlers
- Launch External Scripts
- JavaScript Links and Calculations

Each exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

# Dynamic Writing and Event Handlers

## Objective

In this exercise, you'll deploy some document.write methods to write browser information on the XHTML page. You will also insert an onload event handler to give your users a warm welcome. You will learn about the basic script structure and where JavaScript translates to the browser screen. In addition, you will see when the onload event fires.

## Things to Consider

- Note that JavaScript placement defines where dynamic content is written.
- The navigator object will return information about the browser being used to view the page.
- On event handlers within XHTML tags, use single quotes when placing a string within the events double quotes: onload="show('a string')"
- Be sure to hide your script from the older browsers!

## Step-by-Step Instructions

1. Open the **HelloWorld.html** file in this lab's directory.
2. Within the boundaries of the beginning <body> tag, add an onload event handler that will fire a window.alert method with a message to the user.

```
<body onload="alert('onload event: Hello!');">
```

3. Just below the body tag, add a <script> block. In the beginning <script> tag, do not forget to set the language and type attributes (most browsers do not require the type attribute).

```
<script language="JavaScript" type="text/javascript">
</script>
```

4. In between the <script> tags, add comments to hide the JavaScript from older browsers.

```
<script language="JavaScript" type="text/javascript">  
    <!-- Begin: Hide from old browsers  
  
    // End: Hide from old browsers -->  
</script>
```

5. Between the comments from Step 4, add a document.write statement to write the text <b>Your browser name is: </b>, followed by the name of the browser, which you can get using the navigator object's appName method (remember to use dot syntax). Remember to add an XHTML line break tag at the end of the string so that the output contains a carriage return.

```
<!-- Begin: Hide from old browsers  
document.write("<b>Your browser name is: </b>" +  
    navigator.appName + "<br/>");  
// End: Hide from old browsers -->
```

6. On the next line add another document.write method to display the message <b>Your browser version is: </b> followed by the browser version using navigator.appVersion, and ending with an XHTML line break tag.

```
<!-- Begin: Hide from old browsers  
document.write("<b>Your browser name is: </b>" +  
    navigator.appName + "<br/>");  
document.write("<b>Your browser version is: </b>" +  
    navigator.appVersion + "<br/>");  
// End: Hide from old browsers -->
```

7. After the document.write from Step 6, add one more document.write to show the message <b>Current URL: </b> followed by the URL for the page, and two XHTML line break tags.

```

<!-- Begin: Hide from old browsers
document.write("<b>Your browser name is: </b>" +
navigator.appName + "<br/>");
document.write("<b>Your browser version is: </b>" +
navigator.appVersion + "<br/>");
document.write("Current URL: </b>" +
window.location.href + "<br/><br/>" );
// End: Hide from old browsers -->

```

8. To test your work, use Windows Explorer to browse to the location of your HelloWorld.html file, and double-click it to open it in the default browser, or select the file you want to open, click the File menu, and choose **Open With**. You can also open the file directly from within the Mozilla, Internet Explorer, or Opera browsers by pressing **CTRL+O**, and then browsing to the file.
9. If your exercise does not operate correctly, check for errors by trying one of the following options to detect the source of the error:
  - In Mozilla, go to **Tools>>Web Development>>JavaScript Console** to see which errors are listed, as shown in Figure 6.



Figure 6. The JavaScript console in Mozilla.

- If you are testing in Internet Explorer 5.5+, make sure that script errors are turned on by going to **Tools>>Internet Options**, choosing the Advanced tab, and making sure **Display a notification about every script error** is checked. If there is an error, a warning icon will be displayed in the lower left corner of the browser window, as shown in Figure 7. You can double-click this icon to see the error message shown in Figure 8.



Figure 7. Internet Explorer with the Error icon in the lower left corner.

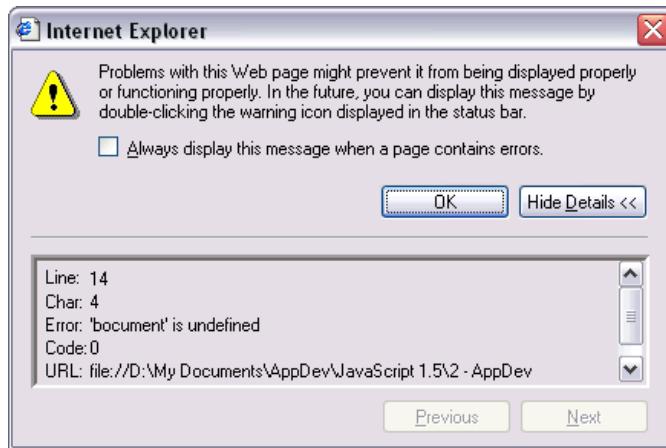


Figure 8. The Internet Explorer error dialog box.

- If you are using Opera 7 +, go to **File>>Preferences** and proceed to the Multimedia area and check **Open JavaScript console on error**. When an error occurs, the error dialog box will pop up as shown in Figure 9.

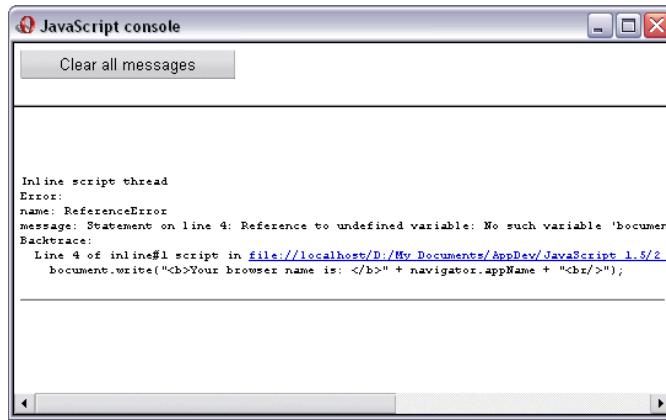


Figure 9. The Opera error dialog box.

# Launch External Scripts

## Objective

In this exercise, you will attach an external library script file to the previous exercises' document.

## Things to Consider

- Although the external JavaScript file does not have to be in the same directory as the HTML document, it will be easier to place them both in the same directory for this example.
- Remember that on a Web site, you would use relative paths in the src attribute of the <script> tag to reference a script in a different directory: src="../scripts/something.js".

## Step-by-Step Instructions

1. Open **HelloWorld.html** from the previous exercise, and create a separate text file in the same directory called **HelloWorld.js**.

**NOTE** In Windows you might not be able to see file extensions. If you don't see files with the .html or .js extensions, choose **Tools>>Folder Options** in the lab folder and then select the View tab. Make sure that **Hide extensions for known file types** is *not* checked.

2. After the <script> block that you added in the first exercise, insert a new <script> block. In the opening <script> tag add an src attribute with the value being the filename of your .js file.

```
<script src="HelloWorld.js" type="text/javascript">  
</script>
```

3. Return to the empty HelloWorld.js file and add the lines required to hide JavaScript from older browsers. Do not add the <script> tags to this file!

```
<!-- Begin: Hide from old browsers
```

```
// End: Hide from old browsers -->
```

4. Add a document.write statement between the hide script lines that says: **<b>Hello from the external script!</b>** and add two line break tags afterwards.

```
<!-- Begin: Hide from old browsers
```

```
document.write('<b>Hello from external  
script!</b><br/><br/>')
```

```
// End: Hide from old browsers -->
```

5. Test the exercise by opening **HelloWorld.html** in your browser. You will see the result from the first exercise and a new message (see Figure 10).

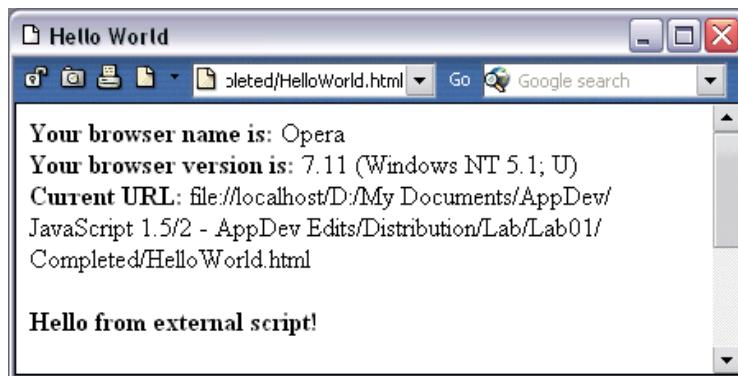


Figure 10. The exercise result in Opera 7.

# JavaScript Links and Calculations

## Objective

In this exercise you will create a link and a button that launches JavaScript. The idea is to learn where you can insert JavaScript for interactivity.

## Things to Consider

- In links or <a> tags, you must use the pseudo-protocol “javascript:” in the href to let the browser know that it should execute the JavaScript function that follows the colon, rather than linking to a URL out on the Internet. This is much like typing http://, which tells the browser you are using the HTTP protocol for browsing.

## Step-by-Step Instructions

1. Open the **HelloWorld.html** file that you updated in the second exercise, if it's not already open.
2. Below the external <script> block, create an unordered XHTML list (<ul>). You can also give the list a type with an attribute, such as **type="disc"**.

```
<script src="HelloWorld.js" type="text/javascript">  
</script>  
  
<ul type="disc">  
  
..</ul>
```

3. For the first line (<li/>) type **From Link:** and then add a link with the text **Say It!** within the link tags. For the links href attribute, launch a window.alert() method that says **Hello from link!**.

```
<ul type="disc">
  <li>
    From Link: <a href="javascript: alert(
      'Hello from link!');">Say It!</a>
  </li>
..</ul>
```

4. After the first `<li>` block, add another `<li>` block and insert a `<form>` block. Set the form's name attribute to “**form1**”.

```
<ul type="disc">
  <li>
    From Link: <a href="javascript: alert(
      'Hello from link!');">Say It!</a>
  </li>
  <li>
    <form name="form1">

    </form>
  </li>
..</ul>
```

5. Between the form tags, type **From Button:** and add a button input tag (`<input type="button">`). Add an onclick event handler attribute (`onclick="<event>"`) to the `<input>` tag to launch a `window.alert()` that says ‘Hello from button!’ when the button is clicked. Also, add a value attribute to the button with the text “**Say It!**”, which will appear as the label on the button.

```
<li>
  <form name="form1">
    From Button:
    <input type="button" value="Say It!" 
      onclick="alert('Hello from button!');"/>
  </form>
</li>
```

6. Under the previous <li> block, add a new <li> block and within it create another form. Set the form's name attribute to “**form2**”.

```
<li>
  <form name="form2">
    </form>
</li>
```

7. Within the form tags from Step 6, create two text input fields (<input type=“text”>). Give them a size attribute of 10 and assign them the names “**number1**”, and “**number2**”, respectively, and place a plus sign between the two text input tags.

```
<li>
  <form name="form2">
    <input type="text" size="10" name="number1"/>
    +
    <input type="text" size="10" name="number2"/>
  </form>
</li>
```

8. After the text input tag named “number2”, add an input button tag with a value attribute set to the equal sign, followed by another text input field with the name attribute set to “**result**”, and the size attribute set to “**10**”:

```
<li>
  <form name="form2">
    <input type="text" size="10" name="number1"/>
    +
    <input type="text" size="10" name="number2"/>
    <input type="button" value="="/>
    <input type="text" size="10" name="result"/>
  </form>
</li>
```

9. Within the “=”input tag, add an onclick event handler. Within this event handler you will use JavaScript to add the values of the “number1” and “number2” input fields, and place the result in the “result” input field.

**NOTE** When adding in JavaScript, you must parse strings into actual integer data types to perform arithmetic calculations. When you retrieve a value directly from an input text element, it is returned as a String, so you must use the parseInt method to make the value an actual number.

```
<li>
  <form name="form2">
    <input type="text" size="10" name="number1"/>
    +
    <input type="text" size="10" name="number2"/>
    <input type="button" value="="
      onclick="document.form2.result.value =
        parseInt(document.form2.number1.value) +
        parseInt(document.form2.number2.value)"/>
    <input type="text" size="10" name="result"/>
  </form>
</li>
```

10. Test the exercise the same way you tested the previous two exercises. Open **HelloWorld.html** in the browser, click on the link and the button in form1 to see their respective actions, and add some numbers together in form2. Figure 11 shows the completed exercise.

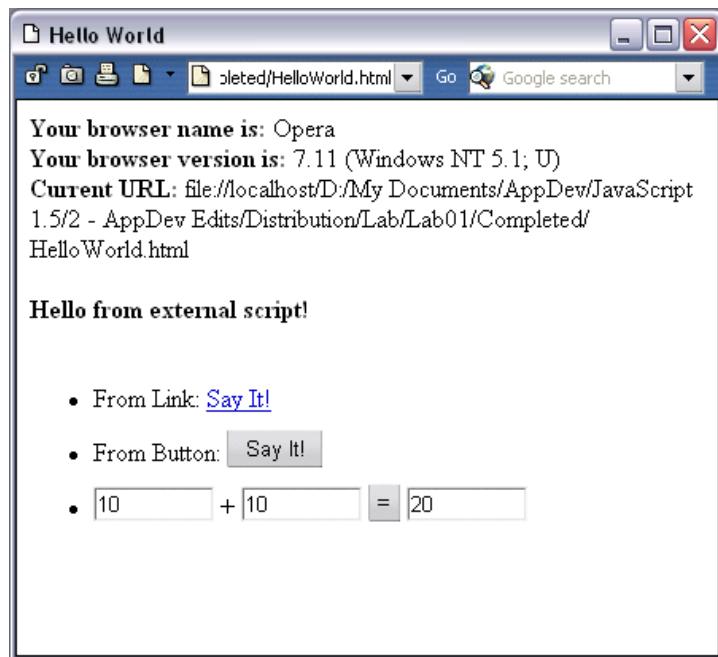


Figure 11. The final result from all of this lab's exercises running in Opera 7.



# JavaScript Conditions and Loops

## Objectives

- Understand how if and if...else constructions work.
- Learn about switch constructions.
- Discover Boolean operators.
- Learn the benefits of looping control structures.
- Minimize typing by using *with*.
- Organize with labeled statements.

Feb 19 2008 3:29PM Dao Dung dungdq@edt.com.vn

For production distribution only – not for distribution or commercial use.

**2-1**

Copyright © 2003 by Application Developers Training Company  
All rights reserved. Reproduction is strictly prohibited.



# The if/if...else Control Structure

You can make simple decisions programmatically by using the if control structure. This control structure is one of the most basic and well-known programming structures. The basic format of an if statement looks like the following:

```
if ([condition]) {  
....// statements to execute if condition is true  
}
```

In English you would read this as “if the condition is evaluated to be true, then execute the statements inside the braces.” If the condition does not evaluate to true, then none of the statements within the braces will be executed. Here is an example of an if statement:

```
var num = 1;  
if ((num + 1) == 2) {  
....alert("The number is 1");  
}
```

You may also want to execute some statements if the condition is false. In this case you would follow your if statement with an else statement:

```
if ([condition]) {  
....// statements to execute if condition is true  
} else {  
....// statements to execute if condition is false  
}
```

While the if/if...else structure is handy for basic code, complex situations will require a more complex construct. If the application must make one hundred decisions, it is very time consuming to write out one hundred individual if statements. Fortunately there are two ways to do this in a much more efficient manner.

## else...if

The else if construct is one way to simplify writing code to handle a number of decisions. Basically, you create an else if statement by joining two if statements with the else keyword:

```
if ([first condition]) {  
    ....// statements to execute if first condition is true  
} else if ([next condition]) {  
    ....// statements to execute if first condition is false,  
        // but next condition is true  
} else {  
    ....// statements to execute if conditions in all the  
        // preceding if statements are false  
}
```

Keep in mind that you can also nest if statements. Nested if statements allow you to handle decision-making when things are more complex, and multiple conditions must be evaluated to determine which statements to execute, as in the following example:

```
if ([parent condition]) {  
    ....// The nested if:  
    ....if ([nested condition]) {  
        ....// statements to execute if parent condition is true,  
            // AND nested condition is also true  
    ....}  
    ....// Another statement in the parent if:  
    ....execute statement;  
    ....  
} else if ([next condition]) {  
    ....// statements to execute if parent condition is false,  
        // but next condition is true  
}
```

The else if structure is handy because you do not have to break out of the structure. Only the relevant statements are executed in a correctly formed else if structure.

You may find that even the else if structure is tedious for a lengthy series of comparisons. Fortunately, there is a quick way to write code to make such decisions in JavaScript.

# The Switch Statement

The switch construct is the quickest, most efficient way to guide your application through a lengthy set of decisions. The switch statement often occurs when the user interface has objects, such as radio buttons, that offer the user several options but only permit one choice.

When a switch statement is declared, it accepts a variable parameter and defines several case constructs. Each case construct represents a possible value of the variable that will be passed into the switch statement. A default case construct is also available for switches.

When the switch receives the variable, it compares the variable's value to each case value until it finds a match. When it finds a matching case value, the statements contained in that case are executed. If no match is found, the switch executes the statements in the default case, if one exists in the code.

```
switch ([variable]) {  
  
    ....case "value1":  
        .....// case 1's statements  
        .....break;  
    ....  
    ....case "value2":  
        .....// case 2's statements  
  
    ....default:  
        .....// These statements will be executed in the event  
        // that no case value matches the variable value  
  
}
```

If there is only one matching case statement, you should insert *break*; as the last statement in the case, which will cause the switch to terminate after executing the statements for that case. Otherwise, the switch will execute the case's statements, and then continue executing the statements for each subsequent case until it encounters a break or reaches the end of the switch.

In the preceding example, if the variable's value matched the first case value, the switch would execute the first case's statements, and the switch would terminate upon reaching the break statement. However, if the variable matched the second case's value, the switch would execute the statements for both the

second case and the default case, because the statements for the second case do not include a break statement.

You can see how much more efficient it would be to write a switch case statement than a series of if...else statements. It is handy to organize your code this way in complicated documents to make the logic easier to understand. Imagine a situation in which each case statement has a different function that it fires as a result, or even that it defines the same function differently as a result. The switch case statement is a powerful, visually appealing way to write decisions.

*See  
**SwitchCase.html***

The following example shows how a switch statement might be used when a user chooses a radio button.

```
<html><head><title>Switch Case Example</title></head>
<body>
....<script language="JavaScript" type="text/javascript">
.....function showDecision() {
.....    var slctRadio;
.....    var msg;
.....    //assign specific form to 'form'.
.....    var form = document.forms[0];
.....    for (i=0; i < 2; i++) {
.....        if (form.rset[i].checked) {
.....            slctRadio = form.rset[i].value;
.....        }
.....    }
.....    switch (slctRadio) {
.....        case "1":
.....            msg = "Selected 1";
.....            break;
.....        case "2":
.....            msg = "Selected 2";
.....            break;
.....        default:
.....            msg = "Not Selected";
.....            break;
.....    }
.....    alert(msg);
.....}
```

```
....</script>
....<form>....
.....<input type="radio" name="rset" value="1"
CHECKED>1
.....<input type="radio" name="rset" value="2">2
.....<input type="button" value="Show"
.....onClick="showDecision()">
....</form>
</body>
</html>
```

# Boolean Operators: NOT/AND/OR and Bitwise

Boolean operators are used to evaluate multiple expressions down to a bottom-line result. One of the most common uses for Boolean math is in search engines. On most search engines' advanced search pages, you will see instructions for Boolean search characters and parameters. This can be very handy for getting the most accurate search possible. In most languages, Boolean math works the same way as in JavaScript. Possible JavaScript Boolean operators are:

- NOT              !
- AND              &&
- OR                ||

## NOT

The NOT operator is used to reverse the Boolean value that follows it. It works as a shortcut to evaluate the opposite of the specified condition. Similarly, in an expression, NOT-Equals (!=) operator is used to test inequality. The following code shows examples of using NOT:

```
!false
// result is true

!(location.href == "http://www.yahoo.com")
// result is false if you are on Yahoo's main page

!((15 / 3) == 5)
// result is false

!(200 > 201)
// result is true
```

If you do not carefully consider the wording of your statement, you may end up with backward logic. The reverse of using the NOT operator is not to use it at all:

```
var bf = false;
var bt = true;
var bn = null;

if (bf) { statement; } // statement will NOT fire
if (bt) { statement; } // statement WILL fire
if (bn) { statement; } // statement will NOT fire
if (undefined) { statement; } // statement will NOT fire
```

Notice that undeclared and null values also evaluate as false.

## AND

The AND operator adds two values to reach a true or false conclusion. AND is the most critical Boolean operator. If either side returns false, the result will be false.

```
(true && true) // true
(true && false) // false
(false && true) // false
(false && false) // false
```

Much different from OR, AND is handy when you need to make sure that two expressions evaluate to true.

## OR

OR is much more forgiving than AND. Basically, if either side is true, the OR operator will evaluate the complete expression to true:

```
(9 > 8 || 7 > 6) // true
(9 > 8 || 7 < 6) // true
(9 < 8 || 7 > 6) // true
(9 < 8 || 7 < 6) // false
if (true == false || false == true || false == false) {
    // any statements here will execute
}
```

## Bitwise Booleans

Bitwise operators are compared at the binary level. This type of comparison is helpful for extremely accurate data types, such as long integers. These would normally be useful in JavaScript only when you are communicating to a higher programming language, via CGI (Common Gateway Interface), servlets, or maybe a Java applet. As soon as the operator has operands to work with, it converts them to a 32-bit binary representation. In addition, both sides of the operator are always evaluated completely before a decision is reached.

**NOTE** Bitwise Booleans is an advanced topic. It is discussed briefly here for completeness.

The following code example illustrates how precise evaluations can use Bitwise Booleans:

```
//Bitwise and
(1.0512312 & 1.01) = 1

//Bitwise or
(1.0512312 > 1.15 | 1.01 < .0015) = 0

//Bitwise xor
1.0512312 > 1.15 ^ 1.01 < .0015 = 0
```

# Loops

In scripting, it will become increasingly important for you to be able to iterate through a series of variables, such as arrays. For example, when you are dealing with forms, the objects are stored in arrays and you may need to cycle through them to look for a particular string.

## for Loops

Loops come in many flavors; the most common is the for loop. The for loop executes a number of statements a specified number of times. In addition, the variable that determines how many times a statement loops is available for use within the for loop. The following example shows the syntax for creating a for loop:

```
for ([initial expr]; [condition]; [update expr]) {  
    // for loop statements to execute  
}
```

You will notice that the parentheses contain three statements separated by semicolons. The first statement, or *initial expression*, normally defines the counter variable. The second statement, or condition, is just like the condition in an if statement; if the condition evaluates as true, the statements in the for loop code block will be executed. The last statement, or update expression, executes at the end of each loop to increment or decrement the counter variable. The following example shows a valid for loop:

```
for (thecounter = 10; thecounter > 0; thecounter--) {  
    document.write("thecounter: " + thecounter + "<br/>");  
}
```

Putting it all together for the previous code snippet, the thecounter variable is initially set to 10. Per the condition statement, while the counter is greater than 0, the for loop will iterate through the code block and execute the document.write statement to display thecounter's current value. At the end of each iteration, the for loop's update statement will subtract 1 from thecounter. After going through the loop 10 times, thecounter will equal 0, which will cause the condition to evaluate as false, and the for loop will terminate.

A handy use for repeat loops is making dynamic select objects in XHTML:

```
document.write("<select name=\"select\">");

for (yr = 1990; yr < 2000; yr++) {
....document.write("<option value=\"" + yr + "\">");
....document.write(yr);
....document.write("</option>");
}

document.write("</select>");
document.close();
```

## Breaking Out of the Loop

Sometimes when you find what you need in the course of the loop, it is more efficient to break out or cancel the loop. This is easy to achieve by using the `break` statement:

```
for (i = 0; i < 10; i++) {
....if (i == 5) {
.....alert("found 5, stopping the loop.");
.....break; //loop will quit.
....}
}
```

In the previous example, the `break` statement executes only when variable `i` is equal to 5. Upon executing the `break` statement, the script breaks out of the `for` loop and begins executing any code after the loop's closing brace.

In the switch case radio button example, you could create an efficient `for` loop by breaking out of the loop when the code encounters the selected radio button.

## Skipping an Iteration

In some circumstances, you might not want the statements in the loop to be executed. If, for example, the user already selected a date from the dynamic

Feb 19 2008 3:29PM Dao Dung dungdq@edt.com.vn

For production evaluation only – not for distribution or commercial use. **JavaScript 1.5**

select object, you might not want to give them the option to choose it from the select object again. In this case, when the loop reaches the date that the user selected, you can use an if statement to determine whether that date has been selected already and skip the statements that add the date to the select object, and finally, use the continue statement to resume the loop.

```
var selectedYr = 1995;
document.write("<select name=\"select\">");

for (yr = 1990; yr < 2000; yr++) {
....if (yr == selectedYr) {
.....continue;
....}
....document.write("<option value=\"" + yr + "\">");
....document.write(yr);
....document.write("</option>");
}

document.write("</select>");
document.close();
```

Now the user's options are narrowed to all but selectedYr.

## while Loops

The while loop is a little different from the for loop, as it can loop for an indefinite period. The while loop does not accept a counter or update expression in the construct, but instead continues to loop until the condition that is provided evaluates to false.

```
while ([condition]) {
    // while loop statements to execute
    ...
    // within the code block, the condition's value must
    // change to something that eventually stops the loop
}
```

As long as the condition is true, the statements within the while loop's code block will continue to execute. Your code should change the value of the

condition variable within the code block to a value that will eventually evaluate as false, in order to stop the loop. It is very easy to create an infinite loop with the while construct, as in the following example:

```
while (true) {  
    window.open("http://www.killpagingfile.com");  
}
```

Obviously, the while loop can be a valuable tool, but you can also see how potentially troublesome a while loop can be. Infinite loops can consume a lot of system memory, and can eventually lead to a computer crash! If an infinite while loop, or even a particularly task-intensive one occurs in Internet Explorer, it will prompt you with “A script is taking longer than expected, continue running the script?” This is actually handy when you are testing loops.

## do–while Loops

The main difference between the while and do–while loops is that the statements within the loop are performed at least once, because they are executed before the condition is evaluated.

```
do {  
    // do-while statements to execute  
} while ([condition])
```

## Using *in* for Property Looping

Often, you may want to loop through the properties of a JavaScript object. You can do this easily with a modified for loop.

```
for ([var] in [object]) {  
    // statements to be executed  
}
```

When the loop executes, var will be assigned the currently iterated property of the loop. For example, if you want to see all the properties of the document object:

Feb 19 2008 3:29PM Dao Dung dungdq@edt.com.vn  
For production evaluation only – not for distribution or commercial use. **JavaScript 1.5**

```
for (var i in document) {  
....document.write("document.");  
....document.write(i + " = " + document[i]);  
....document.write("<br/>");  
}
```

# The with Statement

The with statement is used when you know you are going to access a particular object several times. It was created to save time and energy by allowing you to execute a particular object's methods without referring to the object each time.

```
with (window) {  
....while (true) {  
.....open("http://www.whipeoutmemory.com");  
....}  
}
```

In this case the with statement results in a little more typing, but if you were constantly accessing other objects such as the Date, or Math objects, it would prove useful.

## Labeled Statements

Labeled statements are a neatly typed way of organizing complicated looping constructs. The label itself is just an identifier with a colon defined right before your construct:

```
:  
  
Outer Loop:  
while (true) {  
....// statements to execute  
}
```

Now the loop is labeled. This enables the programmer to explicitly break or continue the labeled loop of choice. For multiple levels of looping, this makes for much more legible code.

```
var d = document;
var rowcount = 20;
var rcounter = 0;
var columns = 10;
var ccounter = 0;

d.write("<table border=\"1\">");

OuterLoop:
while (true) {
....d.write("<tr>");
.....
....InnerLoop:
....while (true) {
.....d.write("<td>Empty Cell</td>");
.....ccounter++;
.....
.....if (ccounter == columns) {
.....ccounter = 0;
.....break InnerLoop;
....}
....}
.....
....d.write("</tr>");
....rcounter++;
.....
....if (rcounter == rowcount) {
.....break OuterLoop;
....}
}

d.write("</table>");
d.close();
```

In this example, a table with 20 rows and ten columns is created out of infinite loops that use labeled break statements. This is not the best way to script this sort of routine, but it serves as a good example.

# Summary

- Add decision making logic to your code by using if, if...else, and switch statements.
- Program more advanced decision-making by using Boolean operators.
- Iteration is a common task in programming and scripting. It is particularly helpful when dealing with arrays.
- The for loop is the best loop to use when you know how many iterations the loop will need to perform.
- While loops are better when you want to loop until a certain condition is met.
- Do-while loops work best if the statements in the loop's code block must be executed at least once.
- With statements save time by allowing you to skip specifying object references every time you wish to execute a method.
- Labeled statements help to organize deeply nested loops by assigning a name to each loop and providing a reference to break or continue any loop within the nesting hierarchy.

# Questions

1. True/False: An if statement always requires a corresponding else block.
2. What part of the if statement must be true for the embedded statements to execute?
3. In a switch statement each \_\_\_\_\_ represents a possible choice.
4. When does the default block in a switch statement execute?
5. True/False:  $(2500 > 1500) \&\& (15 < 3)$
6. True/False: !false
7. What character always follows a loop label?
8. When you use the with statement, an object's \_\_\_\_\_ are easily accessible.
9. Assuming you have an array of 20 options, what is the best loop to make a <SELECT> object out of the array?
10. Which statement allows you to skip the execution of one iteration of a loop and proceed to the next iteration, without breaking out of the loop?
11. Which loop is best if you want the statements inside the loop's code block to execute at least once?

# Answers

1. True/False: An if statement always requires a corresponding else block.  
**False**
2. What part of the if statement must be true for the embedded statements to execute?  
**The condition**
3. In a switch statement each \_\_\_\_\_ represents a possible choice.  
**case**
4. When does the default block in a switch statement execute?  
**When all the cases have been evaluated and none of them match.**
5. True/False:  $(2500 > 1500) \&& (15 < 3)$   
**False**
6. True/False: `!false`  
**True**
7. What character always follows a loop label?  
**: or colon**
8. When you use the with statement, an object's \_\_\_\_\_ are easily accessible.  
**methods**
9. Assuming you have an array of 20 options, what is the best loop to make a <SELECT> object out of the array?  
**The for repeat loop.**
10. Which statement allows you to skip the execution of one iteration of a loop and proceed to the next iteration, without breaking out of the loop?  
**A continue statement.**
11. Which loop is best if you want the statements inside the loop's code block to execute at least once?  
**The do-while repeat loop.**

# Lab 2: JavaScript Conditions and Loops

**TIP:** Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You will find all the code in **TheTroubleWithTableCells.html** and **TheTroubleWithTableCells.js** in the same directory as the sample project. To avoid typing the code, you can cut/paste it from the source files instead.

## Lab 2 Overview

In this lab you will learn about the usefulness of looping and decision constructs. You will also make an advancement in code structuring by following a logical progression to achieve the end result.

To complete this lab, you'll need to work through two exercises:

- Loop the Loop: A Dynamic Table
- Decision Structure: Controlling the Table

Each exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

# Loop the Loop: A Dynamic Table

## Objective

In this exercise, you will use JavaScript to create a dynamic table when the page loads.

## Things to Consider

- The XHTML page is very simple; all it uses is a single external library within the body.
- The main `<table>` tags do not need to be in a loop. They are only used once to make a table.
- The more you can define about the table with variables, and the more you can modularize into separate areas (later, functions), the easier it is to make changes to the script later on down the road.
- Use the escape character `\”` to add a double quote in a string that is defined within double quotes.

## Step-by-Step Instructions

1. Open the **TheTroubleWithTableCells.js** file. The other file, **TheTroubleWithTableCells.html**, is a minimal XHTML document, with a `<script src=>` tag that inserts **TheTroubleWithTableCells.js**, which allows you to concentrate on the JavaScript code without concerning yourself with the HTML that renders the rest of the page.

**TIP:** It is helpful if you can picture the layout of an XHTML table in your mind to help you visualize the tags used to organize a table, which will help you to more easily write the JavaScript code to generate it dynamically. You will recall that the four main elements of a table are the table definition tags (`<TABLE></TABLE>`), the row tags (`<TR></TR>`) that are used to define the rows within the table, and the header tags (`<TH></TH>`) or the cell tags (`<TD></TD>`), which are used to define the columns that appear within each row.

**Lab 2:**  
*JavaScript Conditions and Loops*

---

2. First create the following variables in your .js file to hold values of the table using **var: tblCols = 10, tblRows = 10, tblBorder = 10, tblSpacing = 10, currRowCount = 0** and also add this line at the bottom: **var d = document;** This will save you time when writing the table to the browser—instead of typing **document.write("")** you will only have to type **d.write("")** because you are assigning the document object to the variable ‘d’.

```
//Declare table setup variables.  
var tblCols = 10;  
var tblRows = 10;  
var tblBorder = 10;  
var tblSpacing = 10;  
var currRowCount = 0;  
var d = document;
```

3. Now create an eternal while loop that contains only the boolean keyword **true** in the condition statement. Label the loop **MainLoop:** You want this loop to stop when it has generated the pre-defined number of rows. To accomplish this, put a few blank spaces within the loop, and add the statement **currRowCount++** to increment the count by one every time the loop iterates.

```
MainLoop:  
while (true) {  
    //Empty Line  
    currRowCount++  
}
```

4. Within the while loop, construct an if statement with the condition: **currRowCount == tblRows**.

```
MainLoop:  
while (true) {  
    if (currRowCount == tblRows) {  
  
    }  
    currRowCount++  
}
```

5. Within the if statement add the following line: **break MainLoop;**. This tells the code to stop executing MainLoop once the current row count equals the maximum number of table rows you defined in the variable `tblRows` in Step 2 of this exercise.

```
MainLoop:  
while (true) {  
    if (currRowCount == tblRows) {  
        // Number of rows reached, break infinite loop  
        break MainLoop;  
    }  
    currRowCount++  
}
```

6. With the basic setup complete, it is time to add the guts of the table. Right before the `MainLoop` label, add a line that outputs the beginning `<table>` tag with the basic table attributes set to the variables that were defined in Step 2. Now, go to the line immediately following the closing bracket of the while loop, and add the closing tag for the table. Remember that using `\`` is the equivalent of inserting double quotes within a string that is defined with double quotes.

```
d.write("<table border=\"" + tblBorder + "\\"  
       cellspacing=\"" + tblSpacing + "\"\">>");
```

```
MainLoop:  
while (true) {  
    if (currRowCount == tblRows) {  
        // Number of rows reached, break infinite loop  
        break MainLoop;  
    }  
    currRowCount++  
}  
  
d.write("</table>");
```

**NOTE** Notice that the table tags are being defined around MainLoop. This is because the loop itself will be used to create each row of the table. While there is only one table, there could potentially be many rows within the table (which must be generated in one loop within MainLoop) and many table cells within each table row (which must be generated in another loop within MainLoop).

7. On the first line inside the while loop block, before the if statement, create a d.write statement to output a beginning row tag. Add a blank line after that, and then another d.write tag that writes a closing row tag:

```
while (true) {  
    d.write("<tr>");  
  
    d.write("</tr>");  
  
    if (currRowCount == tblRows) {  
        // Number of rows reached, break infinite loop  
        break MainLoop;  
    }  
    currRowCount++;  
}
```

8. Next, you'll need to insert code to build the individual table cells within the boundary of the row tags. Each row will contain the number of cells defined in the variable named tblCols, which you assigned a value of 10 in Step 2 of this exercise. You can use the tblCols variable in a for loop within the while loop to write out the cells for each row. Insert a new line after the opening <tr> tag and before the ending </tr> tag, and add a for loop with the condition **var tc = 0; tc < tblCols; tc++**:

```
while (true) {  
    d.write("<tr>");  
    for (var tc = 0; tc < tblCols; tc++) {  
  
    }  
    d.write("</tr>");
```

9. On the first line within the for loop, use d.write to output a <td> tag with center alignment, and on the next line insert some empty content into the cell(s) by outputting an empty space. Finally use d.write to generate the closing </td> tag.

```
while (true) {
    d.write("<tr>");
    for (var tc = 0; tc < tblCols; tc++) {
        d.write("<td align=\"center\">");
        d.write(" ");
        d.write("</td>");
    }
    d.write("</tr>");
```

10. On the last line of the file, right after the </table> tag is output, close the document stream by calling d.close().

```
d.write("<table border=\"" + tblBorder + "\""
       cellspacing=\"" + tblSpacing + "\">\n");
```

```
MainLoop:
while (true) {
    if (currRowCount == tblRows) {
        // Number of rows reached, break infinite loop
        break MainLoop;
    }
    currRowCount++
}

d.write("</table>");
```

**d.close();**

11. Now it's time to test your code! Save all of your changes, and run the XHTML file in your browser. If everything goes as planned, you should have a dynamic table with a border of ten pixels, cellspacing at ten pixels, ten rows, and ten columns (see Figure 1).

**Lab 2:**  
*JavaScript Conditions and Loops*

---

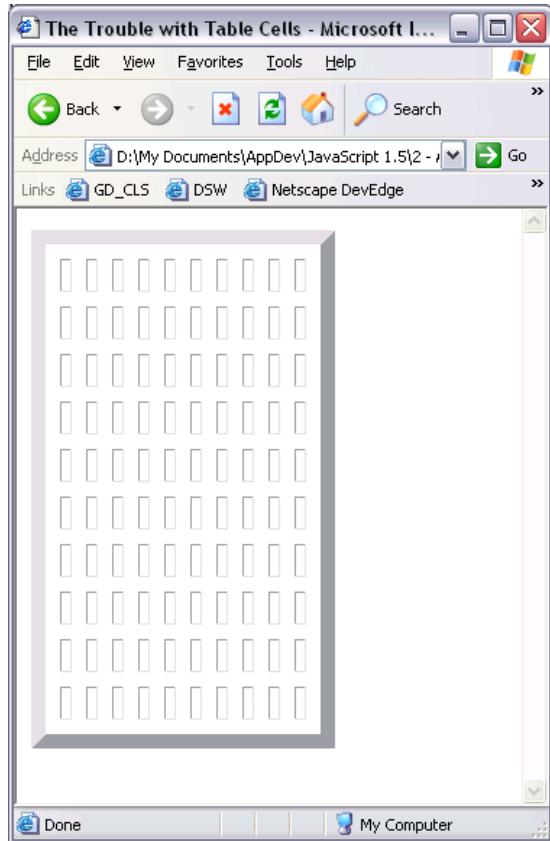


Figure 1. The final result of The Trouble with Table Cells in Internet Explorer.

# Decision Structure: Controlling the Table

## Objective

In this exercise, you will use the easily modifiable structure of the table to do some custom handling. Using the provided objects you will create a table that is different every time the page is hit.

## Things to Consider

- The methods of an object being used in the with statement will only work by themselves within the with block.
- The <TH> tag works just like a regular table cell <TD> tag.

## Step-by-Step Instructions

1. Open **TheTroubleWithTableCells.js** file that you used in the first exercise. This exercise will build on top of the code already in place.
2. Add the following variables to your variable list at the top of the script: hasHeader, hasBground, and currSeconds.

```
//Declare table setup variables.  
var tblCols = 10;  
var tblRows = 10;  
var tblBorder = 10;  
var tblSpacing = 10;  
var currRowCount = 0;  
var d = document;  
var hasHeader;  
var hasBground;  
var currSeconds;
```

*Lab 2:  
JavaScript Conditions and Loops*

---

3. After the new variables, you are going to build a switch decision structure to provide the table with three options for table display. The options are case: 0, case: 1 and case: 2.

```
var hasHeader;  
var hasBground;  
var currSeconds;  
switch() {  
    case 0:  
        break;  
    case 1:  
        break;  
    case 2:  
        break;  
}
```

4. In case 0, add the following statements: **hasHeader = true;** **hasBground = false;** and **break;**. For case 1: add the statements **hasHeader = false;**, **hasBground = false;** and **break;**. And finally, in case 2: add the statements **hasHeader = true;** **hasBground = true;** and **break;**. This defines the three different variations of how the table will be displayed.

```
var hasHeader;  
var hasBground;  
var currSeconds;  
switch() {  
    case 0:  
        hasHeader = true;  
        hasBground = false;  
        break;  
    case 1:  
        hasHeader = false;  
        hasBground = false;  
        break;
```

```
case 2:  
    hasHeader = true;  
    hasBground = true;  
    break;  
}
```

5. Surround the switch statement in a with statement. On the line before the with statement, add the following: **var currDate = new Date();** This will create a new date object with the current date and time. For the condition of the with statement, use **currDate** by itself. Now, for the condition of the switch statement you will use **currDate's getSeconds()** method and modulus: **switch(getSeconds() % 3)**. The results of the **getSeconds()** / modulus condition will be either 0, 1, or 2 depending on what **getSeconds()** returns when this script is run. When **currDate** is defined to **new Date()** the time that is set is NOW, or the exact system time on the computer when the script executes.

**NOTE** You will use two new objects in this step that have not been covered: the **Date()** object and the Modulus operator of the **Math** object. The **Date** object provides you with current date and time information and allows a basic definable object to handle custom dates. The modulus operator, or **%**, simply divides one value by another value and returns the remainder; so the expression **10 % 2** would return 0, while **5 % 2** would return 1.

```
var currDate = new Date();
with(currDate) {
    switch(getSeconds() % 3) { // returns either 0, 1 or 2
        case 0:
            hasHeader = true;
            hasBground = false;
            break;
        case 1:
            hasHeader = false;
            hasBground = false;
            break;
        case 2:
            hasHeader = true;
            hasBground = true;
            break;
    }
}
```

Now, to make things interesting, you'll add code to dynamically set the number of rows and columns displayed in the table, as well as to change the contents of its cells.

6. Immediately following the switch statement block, but within the with block, add the following definitions: **tblRows = getDate();** which overrides default **tblRows** value, **tblCols = getSeconds();** which overrides default **tblCols** value, and finally, **currSeconds = getSeconds();** The **getDate()**, and **getSeconds()** methods will be called from the **currDate** Date object, as defined by the with condition.

```
var currDate = new Date();
with(currDate) {
    switch(getSeconds() % 3) {
        case 0:
            hasHeader = true;
            hasBground = false;
            break;
        case 1:
            hasHeader = false;
            hasBground = false;
            break;
        case 2:
            hasHeader = true;
            hasBground = true;
            break;
    }
}

tblRows = getDate();
tblCols = getSeconds();
currSeconds = getSeconds();
}
```

7. At this point, the table should generate a different number of rows and columns, and different cell contents, depending on the point in time that the browser executes the script. To test it, load **TheTroubleWithTableCells.html** into your browser, and hit the **Refresh** button several times to see if the number of rows and the contents of each cell displayed in the table changes when the page reloads.

In the remaining steps you will generate some additional dynamic variations, and then implement code to change the look of the table based on a factor of the time at which the page is rendered.

Using the switch statement in MainLoop, you will add code for a header row for the table that determines whether the header is displayed, and what background color it has. This has several implications that you must account for in your MainLoop code. First, a header row has to be created before all the other rows without affecting the while loop. Since the header row only appears once in the table, you must detect whether the code is executing the first iteration through the loop so that the header is not repeated. Second, your code must determine the background color to use for the header row. The

**Lab 2:**  
*JavaScript Conditions and Loops*

---

hasBground variable will be used to determine whether the header has a different colored background from the rest of the rows.

8. To get started, you need to add an if/else statement directly after the starting row tag has been generated. The if condition determines whether to generate a header row, so the code must make sure that this is the first iteration of the loop and that hasHeader is true. The condition for the new if statement will be **currRowCount == 0 && hasHeader**.

```
d.write("<tr>");  
if (currRowCount == 0 && hasHeader) {  
  
} else {  
  
}  
d.write("</tr>");
```

9. For the else statement, copy the for loop code from the previous exercise to create the main table cells within the else block. This ensures that the body of table cells will be built, regardless of whether a header row is generated.

```
d.write("<tr>");  
if (currRowCount == 0 && hasHeader) {  
  
} else {  
    for (var tc = 0; tc < tblCols; tc++) {  
        d.write("<td align=\"center\">");  
        d.write("&nbsp;");  
        d.write("</td>");  
    }  
}  
d.write("</tr>");
```

10. Next, since the number of cells in the header will be the same as the column count, you will need to insert a for loop within the if statement using the condition: **for(var tc = 0; tc < tblCols; tc++)**. Within the for loop block, write out the opening header tag: **d.write("<th bgcolor=""")**; On the next line, close the header tag with: **d.write("">Col " + (tc + 1) + "</font></th>")**; In addition, your code should format the contents of each header cell as **Col 1, Col 2**, and so on.

**NOTE** The second d.write statement in Step 10 should include a </font> tag. This will close the statement you'll add in a subsequent step to determine the appropriate font color based on the dynamically selected background color.

```
d.write("<tr>");
if (currRowCount == 0 && hasHeader) {
    for (var tc = 0; tc < tblCols; tc++) {
        d.write("<th bgcolor=\"\"");
        d.write("\">>Col " + (tc + 1) + "</font></th>");
    }
} else {
}
d.write("</tr>");
```

11. Next, you'll need to add code to determine the background color to use for the header row if hasBground is true. Within the two d.write() output statements added in the previous step, add one line: **d.write("#000000");**. This will insert black as the bground color. The beginning header tag must be closed properly, so on the line after the if block add: **d.write("\n align=\"center\"><font color=\"\"");**. This will end the header opening tag, and start the font settings for contents within the header.

```
if (currRowCount == 0 && hasHeader) {
    for (var tc = 0; tc < tblCols; tc++) {
        d.write("<th bgcolor=\"\"");
        if (hasBground) {
            d.write("#000000");
        }
        d.write("\n align=\"center\"><font color=\"\"");

        d.write("\">>Col " + (tc + 1) + "</font></th>");
    }
} else {
```

12. On the line after the d.write() statement that follows the if block in Step 11, create a new if block with the condition: **if (hasBground)**. Within the new if block, add the statement: **d.write("#FFFFFF");** to change the font

**Lab 2:**  
*JavaScript Conditions and Loops*

---

tag's color attribute to white in the event that the header row's background color is black.

```
if (currRowCount == 0 && hasHeader) {  
    for (var tc = 0; tc < tblCols; tc++) {  
        d.write("<th bgcolor=\"\"");  
        if (hasBground) {  
            d.write("#000000");  
        }  
  
        d.write("\" align=\"center\"><font color=\"\"");  
        if (hasBground) {  
            d.write("#FFFFFF");  
        }  
        d.write("\">Col " + (tc + 1) + "</font></th>");  
    }  
} else {  
}
```

The header is now nearly complete; there are only two things that must be fixed in order to finish it.

13. First, since only one header should be created, the hasHeader variable must be set to **false** after the header is created to avoid repeating the header row on subsequent iterations through MainLoop. You must add this code as the last line in the hasHeader if statement:

```
if (currRowCount == 0 && hasHeader) {  
    for (var tc = 0; tc < tblCols; tc++) {  
        d.write("<th bgcolor=\"\"");  
        if (hasBground) {  
            d.write("#000000");  
        }  
  
        d.write("\" align=\"center\"><font color=\"\"");  
        if (hasBground) {  
            d.write("#FFFFFF");  
        }
```

```
        }
        d.write(">Col " + (tc + 1) + "</font></th>");
    } // end of hasHeader if statement

    hasHeader = false;
} else {
}
```

14. Second, since the header row will take up one of the row iterations, the actual number of regular rows will be off by one. To avoid this problem, the loop must be restarted before the row count is incremented. Remember that adding a continue statement within a loop skips over the rest of the code in the current iteration of the loop block, and continues with the next iteration of the loop. A continue statement is perfect for this application, because it will skip over the line of code that increments the currCount variable when the header is complete. Simply add the line **continue MainLoop;** right after the statement that sets the value of the hasHeader variable to false.

```
hasHeader = false;
continue MainLoop;
} else {
}
```

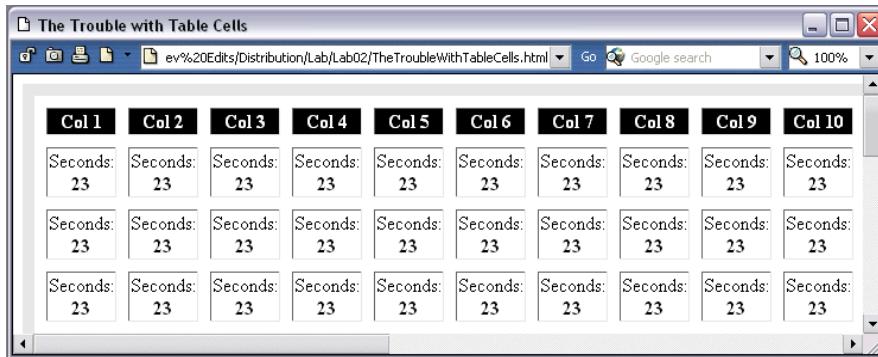
15. The only thing missing is the content for the body <td> cells. On the line in the for loop where you added the non-breaking spaces between the <td> tags in the first exercise, replace **&ampnbsp** with: “**Seconds:<br/><b>**” + **currSeconds + “</b>”**. This will print the word **Seconds:** followed by the number of seconds of the current minute in bold text within each body table cell.

```
} else {
    for (var tc = 0; tc < tblCols; tc++) {
        d.write("<td align=\"center\">");
        d.write("Seconds:<br/><b>" + currSeconds + "</b>");
        d.write("</td>");
    }
}
```

**Lab 2:**  
*JavaScript Conditions and Loops*

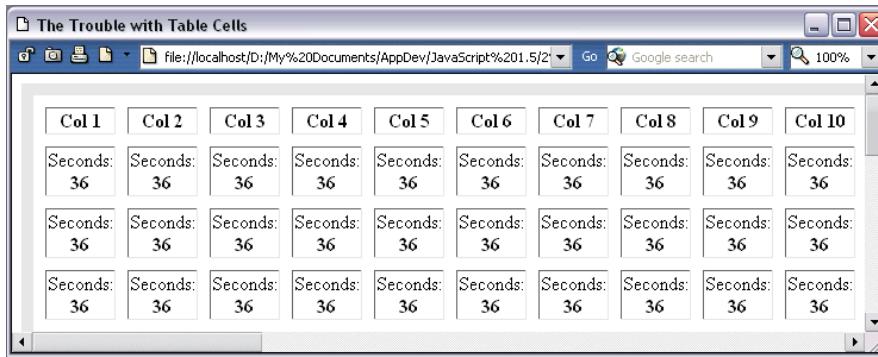
---

16. Now it is time to test your code. Run the **TheTroubleWithTableCells.html** and refresh the page several times. Over time, you should see three different looks to the table as shown in Figure 2, Figure 3, and Figure 4. In addition, there will be a different number of columns and rows each time.



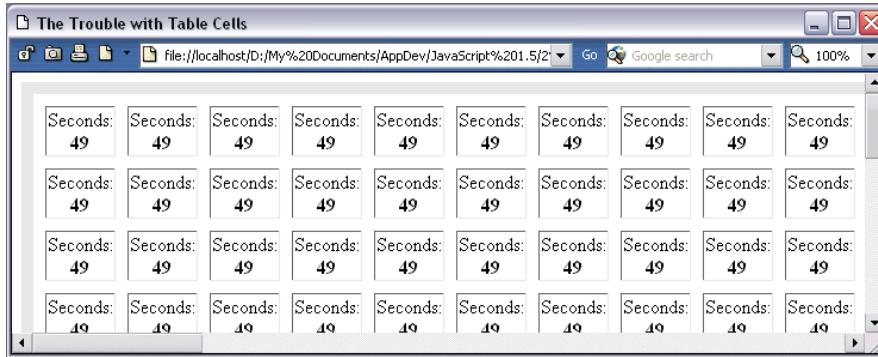
Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	Col 9	Col 10
Seconds: 23									
Seconds: 23									
Seconds: 23									

Figure 2. The table with a header, and with background color.



Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	Col 9	Col 10
Seconds: 36									
Seconds: 36									
Seconds: 36									

Figure 3. The table with a header, and with no background or text color.



Seconds: 49									
Seconds: 49									
Seconds: 49									
Seconds: 49									

Figure 4. The table without a header.

# Strings and Functions

## Objectives

- Learn how to manipulate strings in JavaScript.
- Create your own functions in JavaScript.
- Understand how function parameters are used.
- Discover how to return data from your functions in JavaScript.
- Understand variable scope.

# Strings

You are already familiar with the concept of a string, even though you may not realize it. A string can be simply defined as a set of consecutive characters. In JavaScript, a string is any text that is within a pair of quotes, either single quotes or double quotes. You can even nest one string within another, as you will often see in event handlers. In the following example, you'll notice that the alert() method requires a quoted string as a parameter, while the entire method call must also be within quotes to conform with the HTML standard:

```
onClick = "alert('This is a String.')";
```

You have two ways to assign a string value to a variable. The first is a basic assignment statement:

```
var myString = "hello";
```

The second way to assign a string value to a variable is to create a string object. You do this by using the keyword new followed by a string constructor function:

```
var myString = new String("hello");
```

In the preceding example, the script “constructed” a new string object and gave it the initial value “hello”. Regardless of which way you choose to initialize your variable with a string, the variable that receives the string assignment can respond to all string object methods.

## Manipulating Strings

The ability to manipulate strings is essential for scripters who wish to provide dynamic content on their pages. To aid you in this task, the string object provides many methods that allow you to manipulate strings.

## String Concatenation

Joining two strings to form one string is called *concatenation*. You can perform string concatenation by using one of two JavaScript operators: the addition operator (+) and the add-by-value operator (+=).

The addition operator can be especially useful when you want to link multiple strings to produce text dynamically for your Web page. For example, the following statement joins a string literal to the text value contained in the navigator.appName property and prints a message to the user indicating which browser they are using to view the page:

```
document.write("You are viewing this page with " +  
navigator.appName);
```

While the addition operator can aid you in the concatenation of smaller strings, it can become quite tedious when you want to assemble larger strings into one variable. For example:

```
var msg = "Now is the time";  
msg = msg + " for all good men";  
msg = msg + " to come to the aid";  
msg = msg + " of their country.;"
```

In the preceding examples you'll note that the addition operator allows you to concatenate the string literal with the current value of the variable msg, and assign the result back to the msg variable. While this is a perfectly legitimate way of joining these strings, typing the variable name repeatedly can also be quite tedious and can introduce errors. The use of the add-by-value operator shortens the preceding statements as follows:

```
var msg = "Now is the time";  
msg += "for all good men";  
msg += "to come to the aid";  
msg += "of their country.;"
```

The add-by-value operator allows you to simply append the string literal to msg by combining the concatenation and assignment into one operation.

**NOTE** While JavaScript does not impose a limit on the number of characters in a string, some older browsers impose a limit of 255 characters for a script statement. It is advisable to use the preceding techniques when dealing with excessively large strings, in order to ensure backward compatibility.

**TIP:** You can reference the string object's length property to determine the number of characters contained in the string.

## Changing String Case

The string object offers you two methods to change the case of a string: toUpperCase() and toLowerCase(). Both will change the case of all characters in the string. These methods can be especially useful when comparing two strings that may not have the same case, such as in user input. The following example illustrates this point:

```
var userString = "hello";
var pageString = "Hello";
var StringMatch = false;
if (userString.toUpperCase() == pageString.toUpperCase())
{
    ....StringMatch = true;
}
```

## Substring Searches

You can determine whether one string is contained within another by using the indexOf() method. If you pass a substring into the method as a parameter, this method will return a number indicating the zero-based index position of the character in the parent string where the substring you are searching for begins. If no match occurs, the return value will be -1.

**WARNING!** All indexes accepted by string methods are zero-based, which means that the first character in the string is at position 0 and the last character is at position (string.length-1). If you try to access a character at position string.length, the method won't return a character. This is one of the most common mistakes that new scripters make.

In the following example, suppose you are targeting users of the Netscape Navigator browser, and you want to know whether the user is running a Windows operating system. By querying the navigator.userAgent property,

Feb 19 2008 3:29PM Dao Dung dungdq@edt.com.vn

For product evaluation only – not for distribution or commercial use. **JavaScript 1.5**

Copyright © 2003 by Application Developers Training Company  
All rights reserved. Reproduction is strictly prohibited.



you can get a long string that contains a lot of useful information about the browser. However, you still need to search within the string for the specific operating system information you want.

```
var isWindows = false;
if (navigator.userAgent.indexOf("Windows") != -1) {
....isWindows = true;
}
```

Since you only care if the substring “Windows” is a part of the larger string within the navigator.userAgent property, you don’t need to be concerned with the exact index of the substring’s location. The substring can be at any index position, as long as it is not -1.

## Substring Extraction

You can extract a substring from a larger string by using the String.substring() method. This method takes two parameters: the starting and ending positions of the substring that you want to extract.

```
var colors = "red, blue, green";
var firstColor = colors.substring(0,3);
//result: firstColor = "red"
var secondColor = colors.substring(5, 9);
//result: secondColor = "blue"
var thirdColor = colors.substring(11, 16);
//result: thirdColor = "green"
```

<b>WARNING!</b> Note that the character at the ending index position is not included in the substring that the method extracts. This potential pitfall has snared many a new scripter.
--

This method is most useful in conjunction with other string object methods. For example, by combining your newfound knowledge of the substring() method with the indexOf() method, you can extract a portion of a string based on the position of a known character:

```
var userName = "John Smith"
var lastName = userName.substring(userName.indexOf(" ") +
    1, userName.length)
//result: lastName = "Smith"
```

Table 1 lists some other helpful string object methods.

String Method	Description
charAt( <i>index</i> )	Extracts a single character from a string, given the index of the character.
slice( <i>startIndex</i> [, <i>endIndex</i> ])	Extracts a portion of a string, accepting the starting index of the substring and an optional second parameter. This second parameter can be a negative number indicating the offset from the end of the main string.
split("delimiterCharacter" [, <i>limitInteger</i> ])	Splits a string into pieces delimited by a specific character and stores them in an array. An optional second parameter accepts an integer value to limit the number of array elements.
substr( <i>startIndex</i> [, <i>length</i> ])	Extracts a substring from a larger string when passed the starting index, and the number of characters to extract from the point of the starting index.

Table 1. Useful string object methods.

# Functions

A function is a collection of statements that can be invoked by event handlers or by statements elsewhere within your script. Functions enable you to divide complex operations into their logical building blocks, making your code more manageable. More important, functions enable you to reuse your code, saving you time and effort. By keeping the focus of your functions as narrow as possible, you will eventually create a Toolbox of functions that you can apply to common problems throughout your various projects. Why reinvent the wheel every time you need to use one?

Whereas some other languages make a distinction between procedures (which carry out actions) and functions (which carry out actions and return values), JavaScript has only one function construct. A function in JavaScript can return a value to the statement that invoked it, but it is not required to.

## Creating Your Own Functions

The formal syntax for a function in JavaScript is:

```
function functionName([parameter1]...[, parameterN]) {  
    statement[s]  
}
```

This means that a function declaration begins with the `function` keyword, followed by a function name, which adheres to the same restrictions as HTML elements and variable names. It can be followed by any number of optional parameters enclosed in parentheses; the block of statements that the function will execute is specified in curly braces.

Function parameters and return values are covered later in the chapter, but for now you will see how to define a simple function that illustrates the basics of function construction. The following example defines a simple function that alerts the user with a specific message when the function is called, and a subsequent statement actually calls the function:

```
function warnUser() {  
    ....alert("Warning: Something bad just happened!");  
}  
warnUser();
```

You can link the function to an event handler, such as the onClick event of a button:

```
<INPUT TYPE="button" VALUE="Click Here"  
onClick = "warnUser()"
```

**NOTE** In order to call a function within an event handler, the function call must be enclosed in quotes.

## Function Parameters

Function parameters are essentially variables that are defined in the function declaration. The parameter values are initialized within the calling statement, enabling you to pass values from one statement to another through the function call.

You can specify any number of parameters in a function definition, with the understanding that any code that calls the function must pass it the same number of parameters. In addition, the order in which the parameters appear in the function call must match the order in which they are declared in the function declaration.

In the previous example, which did not use parameters, the same message was always presented to the user. If you want to alert the user with a warning that reflects specific conditions in the program, you would either need to write an entirely new function, or extend the function to accept a string parameter that contains the message you want to display:

```
function warnUser(msg) {  
....alert("Warning: " + msg);  
}  
warnUser("Your monitor is on fire!");
```

When the preceding script executes, the user will see the message “Warning: Your monitor is on fire!”

When you call a function that passes a string parameter from an event handler, you must nest the parameter in single quotes within the double-quoted string of the function call:

```
<INPUT TYPE="button" VALUE="Click Here"  
onClick = "warnUser('Your monitor is on fire!')"
```

## Returning Data from Functions

Just as you can pass values into a function through its parameters, you can also pass a value out of a function through the return keyword. This is often useful when you want to perform an operation on a number of parameters and use the result for some purpose, such as plugging values into a mathematical equation.

**NOTE** It is important to understand that while you can pass in any number of parameters, you can only pass one return value out.

The following example defines a function that accepts the total cost of a meal as a parameter, returns the final bill with a 15 percent gratuity added, and prints the result to the browser:

```
function calcTip(totalBill) {  
....return totalBill + (totalBill * .15);  
}  
document.write(calcTip(10));  
//result: 11.5
```

# Variable Scope

Variables defined outside of functions are called *global variables*, while those defined inside a function are called *local variables*. A global variable is so called because it is accessible to all script statements within the *globe*, which in JavaScript is the currently loaded document. A local variable, by contrast, is only accessible by statements within the function in which the variable is defined.

**NOTE** When a page unloads, all global variables defined in that page are erased from memory. If you need a value to persist from one page to another, you must use some other technique to store the value.

Since local variables are only in scope for the duration of the function in which they are declared, it is possible to reuse variable names within a page. However, this practice can often lead to confusion about which local variable is in use at any particular time, which can make bugs difficult to track down.

It is often convenient to reuse variable names for some types of variables, such as for loop counters. This is considered safe, because the counter variable is reinitialized each time it is reused.

# Summary

- A string is a consecutive set of characters that is enclosed in a quote pair.
- Assign a string value though a simple assignment statement or by creating a string object.
- Concatenate strings by using the addition operator and the add-by-value operator.
- Change a string's case by using the `toLowerCase()` and `toUpperCase()` methods.
- Determine whether a substring exists within a larger string by using the `indexOf()` method.
- Extract a substring from a larger string by using the `substring()` method.
- A function is a set of statements that can be executed by invoking the function name.
- Function parameters are variables defined with the function that enable you to pass values into the function.
- Return a value from a function by using the `return` keyword.
- A variable's scope defines its visibility.

# Questions

1. Which operators enable you to concatenate strings?
2. What would you use the indexOf() method for?
3. How do you use the substring() method?
4. What is the purpose of a function?
5. True/False: A function call does not have to pass the same number of parameters as the function definition.
6. How many return values can a function have?
7. True/False: A local variable defined in function showMsg() can be used in function alertUser().

# Answers

1. Which operators allow you to concatenate strings?  
**The addition and the add-by-value operators**
2. What would you use the indexOf() method for?  
**To determine whether a substring exists within a larger string, and the starting index of that string if it does exist.**
3. How do you use the substring() method?  
**To extract a copy of a substring from a larger string by passing it the index of the first character of the substring and the index of the character after the last character of the substring.**
4. What is the purpose of a function?  
**A function executes a predefined set of statements and optionally returns a value.**
5. True/False: A function call does not have to pass the same number of parameters as the function definition.  
**False. A function call's parameter values must match the function's definition.**
6. How many return values can a function have?  
**Just one**
7. True/False: A local variable defined in function showMsg() can be used in function alertUser().  
**False. Local variables can be used only within the function in which they are defined.**

# Lab 3: Strings and Functions

**TIP:** Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You will find all the code in **MortgageCalc.html**, in the same directory as the sample project. To avoid typing the code, you can cut/paste it from the source file instead.

# Lab 3 Overview

In this lab you will learn how to effectively manipulate strings. You'll also learn how to create functions that simplify your code.

To complete this lab, you will need to work through two exercises:

- Build the Page Dynamically
- Create a Function for the Calculations

Each exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

# Build the Page Dynamically

## Objective

In this exercise, you will create a function that dynamically builds an HTML string that represents a series of controls for your page. The function will return this string. You will also pass this function to a document.write() statement, in the body of your page, in order to render the controls on the page.

## Things to Consider

- HTML is just formatted text. Using a JavaScript document method to write a formatted HTML string is no different than defining it within the body of the page.

## Step-by-Step Instructions

1. Open the **MortgageCalc.html** file located in this lab's directory. It is just a simple page with a header.
2. Create a new function called **buildPage()**.

```
<script language="JavaScript" type="text/javascript">
<!--
<!-- Exercise 1: Build the Page Dynamically -->

function buildPage() {
}

//-->
</script>
```

3. Within your function **buildPage()**, define a variable named **html**, and initialize it to an empty string, using the statement **var html = "";**

```
function buildPage() {  
    var html = "";  
}
```

4. Next, you want to create a string within the function that defines the HTML that you want the browser to render on the page. The controls are listed in Table 2. See Figure 1 at the end of this exercise for the final results.

Element	Description
Form	
Table	width = 400, cellspacing = 10
Table header	Caption, “Mortgage Calculator”
Label, w/ textbox	“Loan Amount”
Label, w/ textbox	“Term of Loan”
Label, w/ textbox	“Interest rate” follow textbox w/ “%” label.
Button	“Calculate”
Label, w/ textbox	“Monthly Payment”

Table 2. Elements for this exercise’s form.

5. Append the following code to the html variable defined in the last step. The use of the add-by-value operator (+=) will make this easier. For your convenience, the code is as follows:

**Lab 3:**  
**Strings and Functions**

---

```
html += "<form name=\"form1\"><table width=\"400\" " +
...."cellspacing=\"10\"";
html += "<tr><td align=\"center\""
    colspan="2"><h3>Mortgage" +
    "Calculator</h3><hr/></td></tr>";
html += "<tr><td>Loan Amount</td>";
html += "<td><input type=\"text\""
name="loanAmount"/></td></tr>";
html += "<tr><td>Term of Loan</td>";
html += "<td><input type=\"text\" name=\"loanTerm\"/>" +
    "Years</td></tr>";
html += "<tr><td>Annual Interest Rate</td>";
html += "<td><input type=\"text\" name=\"interestRate\"/>
%</td></tr>";
html += "<tr><td colspan="2">";
html += "<input type=\"button\" value=\"Calculate\""
    onClick=\"";
html += "</td></tr>";
html += "<tr><td>Monthly Payment</td>";
html += "<td><input type=\"text\""
name="monthlyPayment"/></td></tr>";
html += "</table></form>";
```

6. Return the string value from the function using the statement **return html;**

```
html += "<td><input type=\"text\""
name="monthlyPayment"/></td></tr>";
html += "</table></form>";

return html;
```

7. Finally, in the body of the page, between the `<script>` tags that are provided for you, make a call to **document.write()**, passing your new function **buildPage()** as the parameter, using the statement **document.write(buildPage());**

```
<body>
    <script language="JavaScript">
        document.write(buildPage());
    </script>
</body>
```

8. Save the file and test the exercise by launching MortgageCalc.html in a browser: You should see the controls displayed in Figure 1.

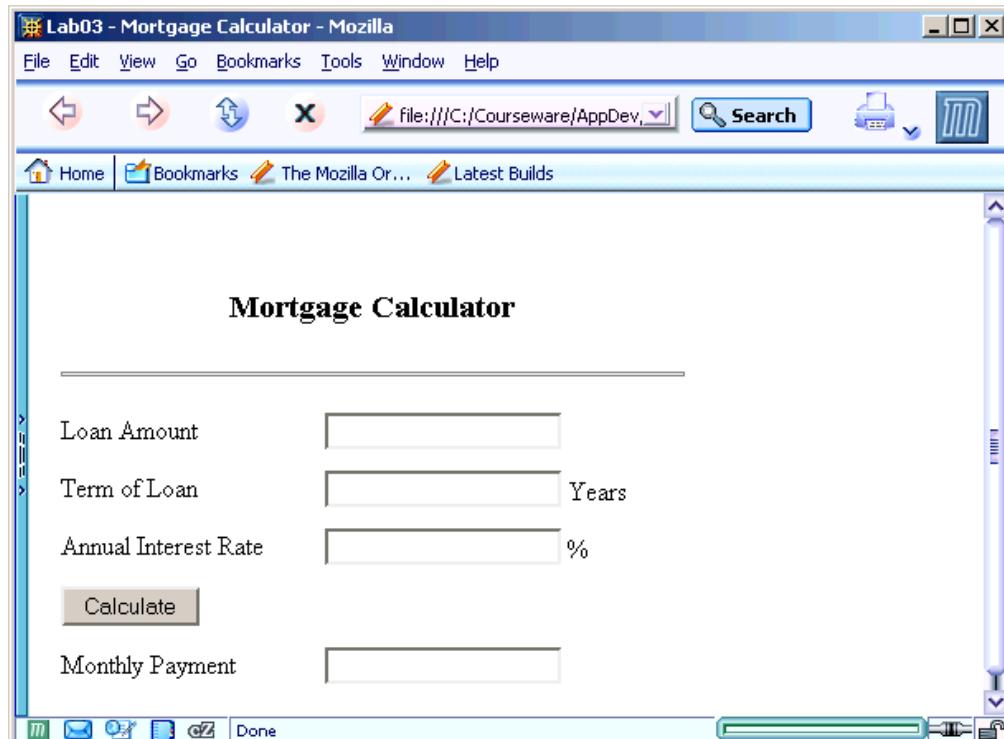


Figure 1. The result of the first exercise.

# Create a Function for the Calculations

## Objective

In this exercise, you'll create a function that calculates monthly payments for a mortgage. Once the function is created, you will hook it to the onClick event of the Calculate button that you created in the last exercise.

## Additional Information

The equation for finding the monthly payments on an amortized loan, is:

$$P = ((r * M) / (1 - (1 + r/n)^{-nt})) / n$$

**P** = the payment, **r** = the annual interest rate, **M** = the mortgage amount, **t** = the number of years, and **n** = the number of payments per year.

**TIP:** You will need to make use of a couple of JavaScript Math object methods that you may not be familiar with. The first method is **Math.pow(value, power)**, which raises a value to a power. The second method is **Math.floor(value)**, which rounds the parameter value down to the next integer less than or equal to itself.

## Step-by-Step Instructions

1. You will be building on the work that you did in the last example, so continue to use the same XHTML file. If you didn't complete the exercise, you can open the **MortgageCalc\_Ex2.html** file in this lab's directory.
2. Create a new function named **calcValues()** under the **buildPage** function from the first exercise.

```
function calcValues() {  
}
```

3. Declare a variable **r**, to represent that interest rate. Initialize variable **r** with the value from the interestRate edit box, divided by 100. This is important, because the input control asks for a percentage, but the calculation requires the decimal equivalent.

```
var r = (document.form1.interestRate.value)/100;
```

4. Declare a variable **M**, to represent the total amount of the mortgage, and initialize the variable with the value from the loanAmount edit box.

```
var M = document.form1.loanAmount.value;
```

5. Declare a variable **t**, to represent the number of years in the loan term. Initialize the variable with the value from the loanTerm edit box.

```
var t = document.form1.loanTerm.value;
```

6. Translate the mortgage equation given in the Additional Information section into a JavaScript statement, and return the value. You will need to make use of the **Math.pow()** method. You can break the statement into multiple statements in order to work with them, but the following code will handle the equation in one statement:

```
return ((r*M) / (1-(Math.pow((1+r/12), (-12*t)))))/12;
```

7. Modify the buildPage() function that you wrote in the first example to call your new function **calcPayments()** within the onClick event for the Calculate button and write the result to the **monthlyPayment** edit box:

```
html += "<input type=\"button\" value=\"Calculate\" " +
"onClick=\"document.form1.monthlyPayment.value=" +
"calcPayments () \"/>";
```

8. Test the exercise at this point to make sure that everything is working. When you enter values into the edit boxes on the page, you should get a float value in the monthlyPayment edit box, with more than two decimal places. To complete the exercise, you should format the value so that it only has two decimal places.

**Lab 3:**  
**Strings and Functions**

---

9. Create a function **formatResult()**, which accepts a parameter named **number**.
10. Write a statement for **formatResult()** that preserves only two decimal places, by multiplying the parameter by 100, dropping all of the decimals of the new value by using the **Math.floor()** method, and dividing the result by 100 in order to return to the original order of magnitude. You should return the following result:

```
return Math.floor(number * 100) / 100;
```

11. Now, modify the return statement in function **calcPayments()** to make use of your new formatting function:

```
return formatResult(((r*M) / (1 - (Math.pow((1+r/12),  
(-12*t)))))/12);
```

12. Test the exercise by launching **MortgageCalc.html** in your browser. The exercise will look like Figure 2.

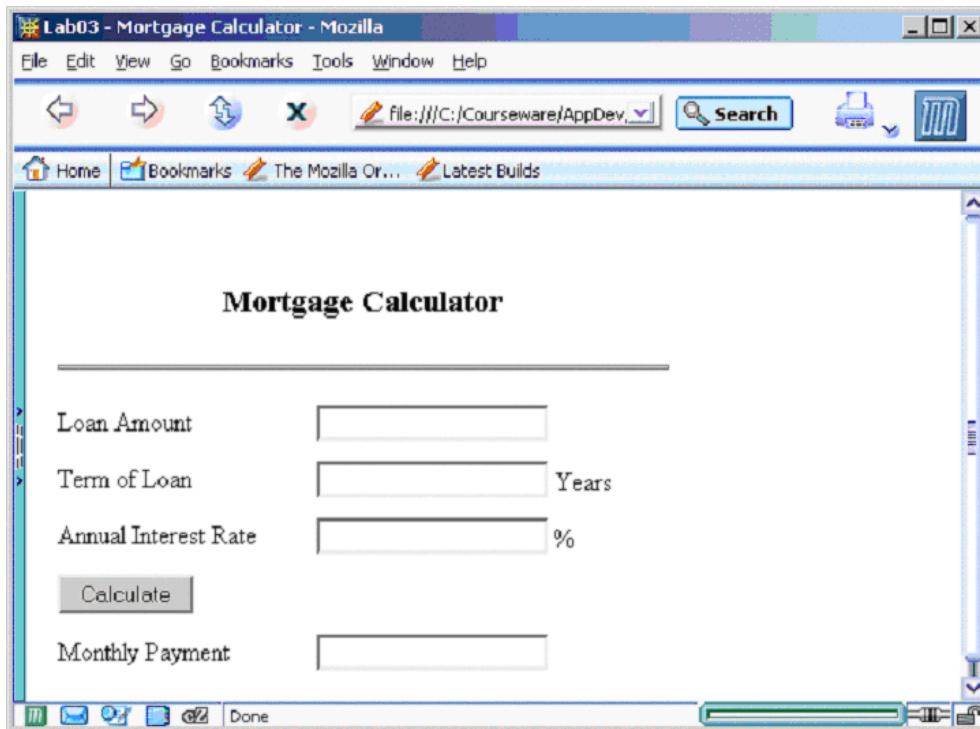


Figure 2. The result of the second exercise.

# Arrays

## Objectives

- Find out the definition of arrays.
- Explore how to use simple arrays.
- Learn the various syntaxes for creating arrays.
- Learn how to create array structures.
- Discover how simple arrays differ from parallel arrays.
- Understand how to use parallel arrays.
- Find out the definition of multidimensional arrays.
- Use multidimensional arrays to manage structures.
- Learn about JavaScript array properties and array methods.
- Use the join and slice array methods.

# Introduction to Arrays

Arrays represent lists of data. They exist in almost every programming language because they solve common problems. Applications frequently need to display lists, but it is best not to create a large number of meaningless variable names. For example, if you need to keep a list of employee names, you could write unique variable names for each employee:

```
var empName1 = "Homer"  
var empName2 = "Carl"  
var empName3 = "Lenny"
```

As you can see, building a long list of employees with this method results in very cumbersome coding. The alternative uses an array to hold the list of employees:

```
var empList = new Array()  
empList[0] = "Homer"  
empList[1] = "Carl"  
empList[2] = "Lenny"
```

The second version offers much more convenience and flexibility.

Arrays are defined as an ordered collection of items. Because of the loose typing in JavaScript, this collection may consist of different types of data. For example, the following example represents a legal array in JavaScript:

```
var things = new Array()  
things[0] = 1;  
things[1] = "Homer"  
things[2] = form.elementsSymbol.selectedIndex
```

In other words, arrays in JavaScript are *heterogeneous*: you may place mixed types within the same array. This differs from other, strongly typed languages like C and Java, which require that all the elements of the array contain the same type of element.

Arrays are the only collection type built into the JavaScript language. Therefore, you use them any time you need to organize a collection of information. You will see that arrays offer a great deal of flexibility in their

declaration and use, mitigating the absence of other collection types. Unlike other languages, which contain numerous collections, arrays suffice for whatever you need in JavaScript.

You can declare arrays in JavaScript in several different ways, partially because of support for arrays in previous versions. With the first method, which comes from JavaScript 1, you create an object constructor to create the array. An object constructor is a function that creates and returns the array object, ready for items.

```
function makeArray(n) {  
    this.length = n  
    return this  
}
```

To create the array in code, use the new keyword.

```
var anArray = new makeArray(10)
```

JavaScript 1.5 introduced a new, more intuitive syntax when it made arrays full-blown objects, like the other objects that exist in the language. Now, you don't have to create your own constructor; you can call the one that the language provides.

```
var newImprovedArray = new Array(10)
```

The array object automatically has a length property, which defaults to zero if you don't specify a size. In the previous code, the array is initialized to a size of 10.

To access an element of an array, you index it starting at zero. All indexing in JavaScript starts at zero. To access the fourth element of an array, use index 3:

```
anArray[3] = "Waylon"
```

JavaScript does not require that all elements have values. For example, you can create an array with ten elements but only place values in a few of the slots. If you access an element that does not have a value in it, the results are undefined. Consider the following code:

```
var anArray = new Array(10);
anArray[5] = 6 // 6th element of the array
              // contains 6
anArray[3]     // undefined
```

In this code, the sixth element (indexed by 5) is assigned a value, making it accessible. However, the fourth element is not defined, so the browser displays the value undefined.

Arrays are objects, meaning that they have intrinsic properties and methods. One of the most important properties is length. You can use this property to determine the number of elements allocated for the array. In the previous code example, the length of the array equals 10, accessible via indexes 0 through 9. This property exists exactly like an array element, available through dot notation.

```
anArray.length // equals 10
```

Arrays, like most of JavaScript, are dynamic, which means that you don't have to resize them if you add more items. JavaScript automatically resizes the array, adding more indexes, to accommodate the elements. In fact, using the length property is a good way to add an element to the end of the existing list of items.

```
anArray[anArray.length] = "New Item"
```

# Simple Arrays

The usefulness of arrays appears often in JavaScript. For example, a simple array can populate an HTML textarea control. Figure 1 shows a list of the elements in an HTML textarea control.

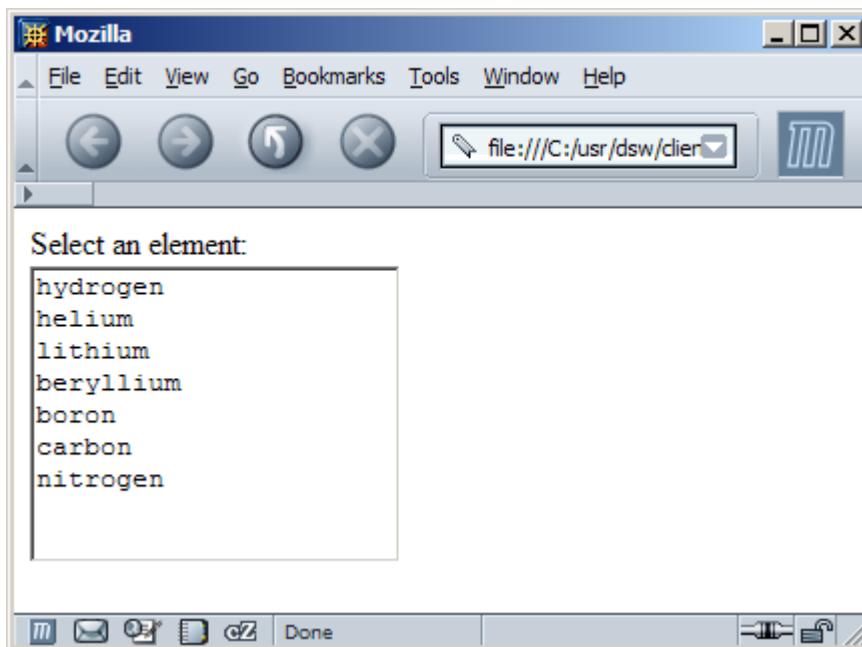


Figure 1. JavaScript generates the list of elements from an array.

See *SimpleArray.js*

The first part of the sample is the JavaScript source file.

```
var elements = new Array(7)
elements[0] = "hydrogen"
elements[1] = "helium"
elements[2] = "lithium"
elements[3] = "beryllium"
elements[4] = "boron"
elements[5] = "carbon"
elements[6] = "nitrogen"
```

```
function generateElements(form) {  
    var elementList = ""  
    for (var i = 0; i < elements.length; i++) {  
        elementList += elements[i] + "\n";  
    }  
    form.textAreaElements.value = elementList;  
}
```

This example builds an array with the first seven elements from the periodic table. It allocates an array of size 7, then populates all the items of the array. The function generateElements() accepts a DOM form object as a parameter, then builds a long string with all the element names, each separated with a newline character ("\n"). Finally, the function sets the value property of the form's textAreaElements control (the text area) to the created list.

*See*  
***SimpleArray.html***

The HTML page that uses this function includes this file at the top of the page:

```
<html>  
<script language="JavaScript" src="SimpleArray.js"/>  
    <body onload="generateElements (document.elementData)">  
        <form name="elementData">  
            Select an element:<br/>  
            <textarea name="textAreaElements" rows="8" cols="20"/>  
        </form>  
    </body>  
</html>
```

This HTML file calls the generateElements method as the page is loaded to populate the textarea.

# Arrays as Structures

The syntax that you saw in the previous section for declaring and using arrays represents only one possibility. As mentioned earlier, arrays in JavaScript are the primary data structure for lists of items. This includes lists that are represented in other data structures in other languages. For example, the C language includes a struct construct, which allows a single type identifier to include numerous fields. Similarly, the “class” structure in Java supports the same kind of operation.

Arrays in JavaScript may act as data structures with individual fields. The primary difference between how JavaScript handles Arrays versus other languages goes back to the loosely typed nature of JavaScript. Ultimately, JavaScript isn’t affected by the type of element you add to the array or if it is predefined. This represents both the good and bad effects of loose typing. Loose typing is convenient because you don’t need to create definitions up front before you use variables. On the other hand, you can accidentally create new variables if you misspell an existing variable name.

You can use arrays in JavaScript to create structured elements. For example, this is an array declaration, although it certainly doesn’t look like the previous examples:

```
var hydrogen = new Array();
hydrogen.symbol = "H"
hydrogen.name = "hydrogen"
hydrogen.weight = 1
```

In this case, the assignment of the array elements uses names instead of numeric indices. In the newer versions of JavaScript (like version 1.5), the manner in which you define the array determines how you access the values. In older versions of JavaScript, you could access elements like this either by numeric value or by name. In JavaScript 1.5, you must access the elements by name if they are defined by name (as above).

Arrays used in this manner enable you to create data structures that have distinct fields. In the next example, the page shows an HTML select that displays the symbol for an element. When the user selects a different element, the display changes dynamically to show the name and weight of that element. The user interface appears in Figure 2.

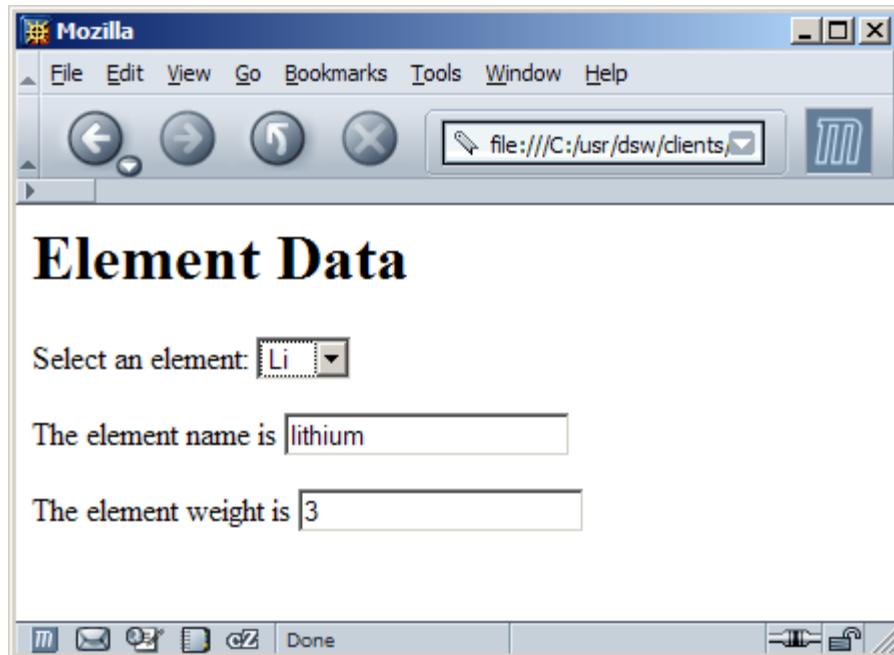


Figure 2. The name and weight populate dynamically when the element symbol changes.

See

**StructureArray.js**

The JavaScript source for this sample is:

```
var hydrogen = new Array();
hydrogen.symbol = "H"
hydrogen.name = "hydrogen"
hydrogen.weight = 1

var helium = new Array();
helium.symbol = "He"
helium.name = "helium"
helium.weight = 2

var lithium = new Array();
lithium.symbol = "Li"
lithium.name = "lithium"
lithium.weight = 3
```

```
var beryllium = new Array();
beryllium.symbol = "Be"
beryllium.name = "beryllium"
beryllium.weight = 4

var boron = new Array();
boron.symbol = "B"
boron.name = "boron"
boron.weight = 5

var carbon = new Array();
carbon.symbol = "C"
carbon.name = "carbon"
carbon.weight = 6

var nitrogen = new Array();
nitrogen.symbol = "N"
nitrogen.name = "nitrogen"
nitrogen.weight = 7

var elements = new Array(hydrogen, helium, lithium,
                        beryllium, boron, carbon, nitrogen)

function generateElements(form) {
    var i = form.elementSymbol.selectedIndex
    form.elementName.value = elements[i].name;
    form.elementWeight.value = elements[i].weight;
}
```

This listing first builds the individual structured items. Each element consists of a symbol, a name, and its atomic weight. Once the element's definition is complete, the array of elements may instantiate. Notice that the elements array is really an array of arrays. Each item in the array is itself another array. JavaScript supports nesting arrays in this manner.

The generateElements() method determines which element symbol the user selected by looking at the form's elementSymbol field (which is the HTML select control), then populates the other two fields on the page with the matching values. The syntax for accessing a particular field in the structured array uses the typical JavaScript dot notation. For example, to access helium's atomic weight, you would use this syntax:

```
form.elementWeight.value = elements[1].weight;
```

This syntax allows you to nest an array inside another array. Even though the inner array looks like a data structure (because all the fields of the array have names), JavaScript stores it as another array.

The structure syntax resembles similar syntax in other programming languages such as C. However, an important distinction exists. Because of the loose typing of JavaScript, a misspelled name creates a new array element with no warning.

See **Structure Array.html**

The following code snippet contains the HTML that renders the Structure Array JavaScript example:

```
<html>
<script language="JavaScript"
src="StructureArray.js"></script>
<body onload="generateElements(document.elementData)">
<h1>Element Data</h1>
<form name="elementData"><p>
Select an element:
<select name="elementSymbol"
onchange="generateElements(this.form)">
<option>H</option>
<option>He</option>
<option>Li</option>
<option>Be</option>
<option>B</option>
<option>C</option>
<option>N</option>
</select></p>
<p>The element name is <input type="text"
name="elementName"/></p>
<p>The element weight is <input type="text"
name="elementWeight"/></p></form>
</body>
</html>
```

# Parallel Arrays

The structured array example in the previous section represents an example of a nested array, or an array defined inside another array. An alternative to a structured array creates two or more arrays that are parallel. In other words, the elements in the second position in all the arrays relate to one another.

As an example, the same “elements” page from the previous section uses parallel arrays for the implementation. The resulting page appears in Figure 3.

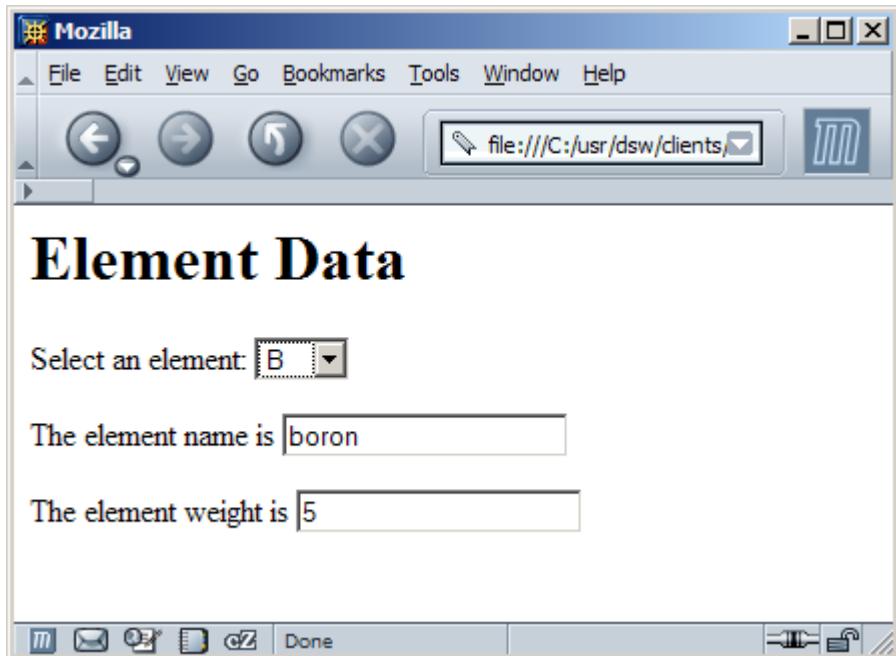


Figure 3. The “elements” page using parallel arrays.

See  
*ParallelArray.js*

The JavaScript file for this page defines the parallel arrays and the `generateElements()` function.

```
var elementName = new Array("hydrogen", "helium",
    "lithium", "beryllium", "boron", "carbon",
    "nitrogen")
var elementSymbol = new Array("H", "He", "Li", "Be",
    "B", "C", "N")
var elementWeight = new Array(1, 2, 3, 4, 5, 6, 7)

function generateElements(form) {
    var i = form.elementSymbol.selectedIndex;
    form.elementName.value = elementName[i];
    form.elementWeight.value = elementWeight[i];
}
```

First, the code creates the array containing the full names of the elements. Next, it creates an array of the corresponding element symbols, followed by another array containing each element's atomic weight. The developer must take great care to reference the same position in each array, to ensure that the code references the correct name, symbol, and atomic weight for any given element. Using parallel arrays can cause debugging problems if the elements are not correctly synchronized.

The generateElements() method performs a familiar task: it discovers the selected symbol from the HTML select control and synchronizes the other arrays to that value as it fills in the values of the other controls.

See

*ParallelArray.html*

The following code creates the HTML page that displays the elements.

```
<html>
<script language="JavaScript" src="ParallelArray.js"/>
<body onload="generateElements(document.elementData)">
<h1>Element Data</h1>
<form name="elementData"><p>
Select an element:
<select name="elementSymbol"
       onChange="generateElements(this.form)">
<option>H</option>
<option>He</option>
<option>Li</option>
<option>Be</option>
<option>B</option>
<option>C</option>
<option>N</option>
</select></p>
<p>The element name is
    <input type="text" name="elementName"/></p>
<p>The element weight is
    <input type="text" name="elementWeight"/></p>
</form>
</body>
</html>
```

As in previous versions, the onload event of the page fires initially to populate the controls. The same method fires in the onChange event of the form to synchronize the elements whenever the select control changes.

This version of the application is much more succinct. The definition of the arrays occupies much less room than the structured version. The synchronization problem is the only downside to this version.

# Multidimensional Arrays

Structured arrays actually utilize the array feature of JavaScript. An array defines an array created inside another array, which is exactly what we did for the structured array example. You can create arrays with as many dimensions as you like in JavaScript, but they become conceptually difficult past about three dimensions. When creating two-dimensional arrays, you may think of the data as resembling a table, with rows and columns. The first dimension of the array represents the rows and the second represents columns.

Multidimensional array definition may use the structured approach shown previously. It may also use syntax more similar to the parallel array example. In this example, we build the elements page by using multidimensional arrays declared in the parallel array syntax. The page that results from this example is the same as the previous versions, and it appears in Figure 4.

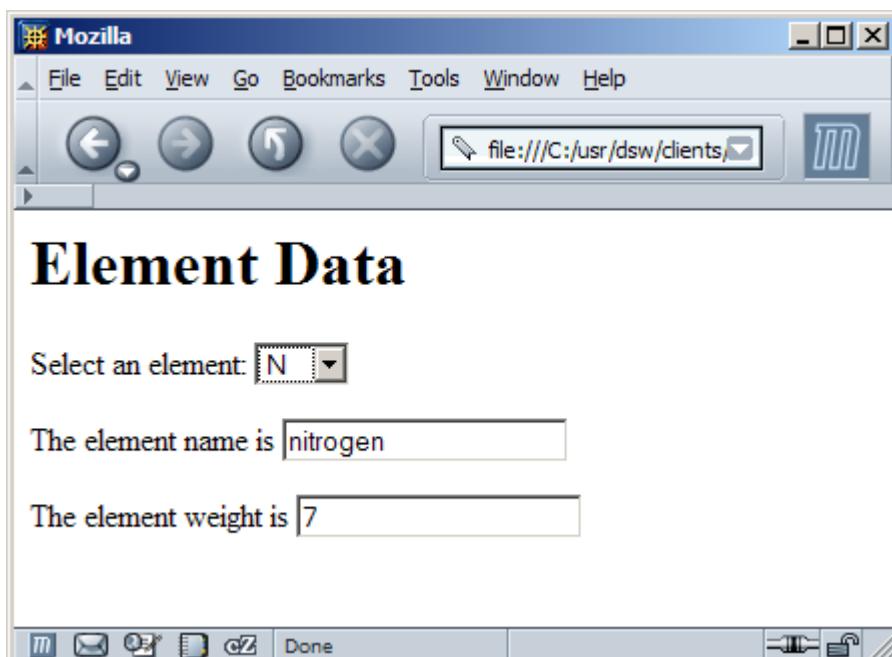


Figure 4. The user interface using multidimensional arrays.

See  
**MultiDimensional  
 Array.js**

The first file in the sample defines the arrays and the generateElements() function.

```
elements = new Array()
elements[0] = new Array("H",      "hydrogen",      1)
elements[1] = new Array("He",     "helium",        2)
elements[2] = new Array("Li",     "lithium",       3)
elements[3] = new Array("Be",     "beryllium",     4)
elements[4] = new Array("B",      "boron",         5)
elements[5] = new Array("C",      "carbon",        6)
elements[6] = new Array("N",      "nitrogen",      7)

function generateElements(form) {
    var i = form.elementSymbol.selectedIndex
    form.elementName.value = elements[i][1]
    form.elementWeight.value = elements[i][2]
}
```

The elements array contains the same information as before, created with a more terse syntax. Each of the elements in the outer elements array includes another array definition for the individual element characteristics.

The generateElements() function performs the same function as before. However, it now uses the more traditional multidimensional array syntax of square brackets to distinguish the array elements. For example, the following line accesses the element in the fifth “row” (starting at 0) of the element table and the third “column”, which results in the value of 5:

```
form.elementName.value = elements[4][2]
```

See

*MultiDimensional  
Array.html*

The HTML user interface for this page is very similar to previous examples:

```
<html>
<script language="JavaScript"
src="MultiDimensionalArray.js"/>
<body onload="generateElements(document.elementData)">
<h1>Element Data</h1>
<form name="elementData"><p>
Select an element:
<select name="elementSymbol"
onchange="generateElements(this.form)">
<option>H</option>
<option>He</option>
<option>Li</option>
<option>Be</option>
<option>B</option>
<option>C</option>
<option>N</option>
</select></p>
<p>The element name is
<input type="text" name="elementName"/></p>
<p>The element weight is
<input type="text" name="elementWeight"/></p>
</form>
</body>
</html>
```

As you can see, virtually the same HTML document supports the array definition in all its possible forms.

# Using the Array Object

Arrays are full-blown objects in JavaScript, meaning that they have properties and methods like other built-in objects. Properties allow you to investigate characteristics of the array (such as length), and methods allow you to execute specific code on the contents of the array. This section shows some of the useful properties and methods.

## Length

Arrays include a length property, which returns the number of entries in the array. The entries encompass any JavaScript type, including null. If you have an array with an entry in the tenth index and all the other entries are null, the length property returns 10.

The length property enables you to add a new element to an existing array without worrying about how many elements already exist. For example, you can add a new element to the end by using the length as the index:

```
anArray[anArray.length] = "New Item"
```

The length property is also useful when programmatically looping through the contents of an array. In the SimpleArray example at the beginning of this chapter, the contents of the array appeared in a textarea control, and the generateElements() function iterated over the array to create the contents.

```
function generateElements(form) {  
    var elementList = ""  
    for (var i = 0; i < elements.length; i++) {  
        elementList += elements[i] + "\n";  
    }  
    form.textAreaElements.value = elementList;  
}
```

## Concat

The concat() method enables you to join two arrays to form a new, larger array. In other words, it concatenates one array to another, returning a new array that contains the contents of both. Consider the following example:

```
var arrayOdd = new Array(1, 3, 5)
var arrayEven = new Array(2, 4, 6)
var arrayMixed = arrayOdd.concat(arrayEven)
// arrayMixed == 1, 3, 5, 2, 4, 6
```

The arrayMixed array contains the contents of the first array concatenated with the contents of the second array.

## Join

Frequently, you need to display the contents of an array as a continuous string. The first example (SimpleArray) in this chapter performed this mapping by using a for loop to iterate over the array, building a string by concatenating the contents of the array one element at a time. The generateElements() method from SimpleArray appears here:

```
function generateElements(form) {
    var elementList = ""
    for (var i = 0; i < elements.length; i++) {
        elementList += elements[i] + "\n";
    }
    form.textAreaElements.value = elementList;
}
```

The need to represent an array as a string is so common that the join() method was introduced to make it easy to perform the same task as the method above. The join() method accepts a single parameter, which represents the delimiter that is used as it creates a string of the contents of the array. Using the join() method, the entire function listed above collapses to a single line of code:

```
function generateElements(form) {
    form.textAreaElements.value = elements.join("\n")
}
```

The join() method accepts the newline character ("\\n") as the delimiter and returns the array as a string, with each element separated by a newline. As you can see, understanding the features that already exist in JavaScript can save a great deal of time.

See **SimpleJoin.js**

The following example illustrates the use of the join method in a JavaScript source file:

```
var elements = new Array(7)
elements[0] = "hydrogen"
elements[1] = "helium"
elements[2] = "lithium"
elements[3] = "beryllium"
elements[4] = "boron"
elements[5] = "carbon"
elements[6] = "nitrogen"

function generateElements(form) {
    form.textAreaElements.value = elements.join("\n")
}
```

See  
**SimpleJoin.html**

The HTML code that follows implements the JavaScript source file for SimpleJoin.js:

```
<html>
<script language="JavaScript"
src="SimpleJoin.js"></script>
<body onload="generateElements (document.elementData)">
<form name="elementData">
    Select an element:<br/>
    <textarea name="textAreaElements" rows="8" cols="20"/>
</form>
</body>
</html>
```

## Slice

Occasionally, you need only a portion of an array. The slice() method enables you to extract a contiguous selection from an existing array. You must specify a starting entry (by index number), and you may optionally specify an ending index. If you do not specify an ending index, JavaScript returns the remainder of the array, starting from the start index.

Consider this example:

```
var elementName = new Array("hydrogen", "helium",
    "lithium", "beryllium", "boron", "carbon",
    "nitrogen")
var simpleElements = elements.slice(0, 2)
// simpleElements == "hydrogen", "helium", "lithium"
```

The slice() method enables the developer to extract the first three elements as a new array. The original array is not affected by this method.

Note that each of the array methods returns the resulting array, so you may chain them together to achieve specific results. For example, you might want to slice an array and then output it as a string. You can write two lines of code to perform this operation:

```
var theSlice = elements.slice(0, 2)
form.textAreaElements.value = theSlice.join("\n")
```

Alternatively, you can compress this to a single line of code:

```
form.textAreaElements.value =
elements.slice(0, 2).join("\n")
```

# Summary

- Arrays represent structured data in list form.
- Arrays are declared using several different syntaxes, based on the JavaScript version in use.
- Simple arrays are created with the Array object constructor.
- Arrays can be used to populate HTML controls such as the textarea control.
- Arrays can mimic structures from other languages.
- Structured arrays enable you to manage disparate types of data in a list.
- Parallel arrays enable you to combine simple array syntax with the functionality of structured arrays.
- Multidimensional arrays define arrays within other arrays.
- Multidimensional arrays can be used to store complex data.
- The array length property returns the number of entries in the array.
- The concat() method concatenates two arrays, returning a longer array.
- The slice() method enables the developer to extract a portion of an array.
- The join() method creates a string with all array elements, using the delimiter that is defined as the parameter.
- The methods of arrays return the array, so you can chain them together.

# Questions

1. True/False: Arrays are the only data collection type in JavaScript.
2. Which JavaScript types are allowed in arrays?
3. What is a structured array?
4. How are parallel arrays used?
5. What is a multidimensional array?
6. Why is the join() method used?
7. Why do you use the slice() method?

# Answers

1. True/False: Arrays are the only data collection type in JavaScript.  
**True. Arrays are flexible enough to serve as the only collection type that you need in JavaScript.**
2. Which JavaScript types are allowed in arrays?  
**All types**
3. What is a structured array?  
**An array in which each of the elements is defined by an array that uses names instead of indices to identify the array elements.**
4. How are parallel arrays used?  
**To represent collections of like data by creating several arrays in which each element at a particular index position relates to the other elements in that position.**
5. What is a multidimensional array?  
**A multidimensional array is one in which each element of the array is also an array.**
6. Why is the join() method used?  
**The join() method returns the entire contents of the array as a string, using the delimiter that is passed as a parameter to separate the items.**
7. Why do you use the slice() method?  
**() To return a portion of an array from the start index to the (optional) end index.**

# Lab 4: Arrays

**TIP:** Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You will find all the code in **JMartSale.html** and **JMartSale.js**, in the same directory as the sample project. To avoid typing the code, you can cut/paste it from the source files instead.

# Lab 4 Overview

In this lab you will learn how to parse delimited strings into Arrays and how to properly dump multidimensional arrays into viewable data.

To complete this lab, you will need to work through two exercises:

- Delimited String to Arrays
- Display the Product

Each exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

# Delimited String to Arrays

## Objective

In this exercise, you will create a function that accepts a comma-delimited string of product categories, an array of delimited strings of products, and a delimiter to split the strings with. The function will return a completed array that will contain an array of arrays that hold the category at index [0] and the products in an array at index [1]. You will also create a quick function to start the array creation.

## Things to Consider

- To access an array that is within an array use the following syntax:  
mainarray[indexinteger][subarrayindexinteger].
- A delimited string is a string of different values separated by some sort of a delimiter, most often a comma: “bob,Ralph,frank,steve”.  
Programmatically these are easy to convert into arrays, or import into spreadsheets and such.
- A three dimensional array is an array packed with arrays that are also packed with arrays.

## Step-by-Step Instructions

1. To get an idea of how a 3D array might look, examine Figure 5.

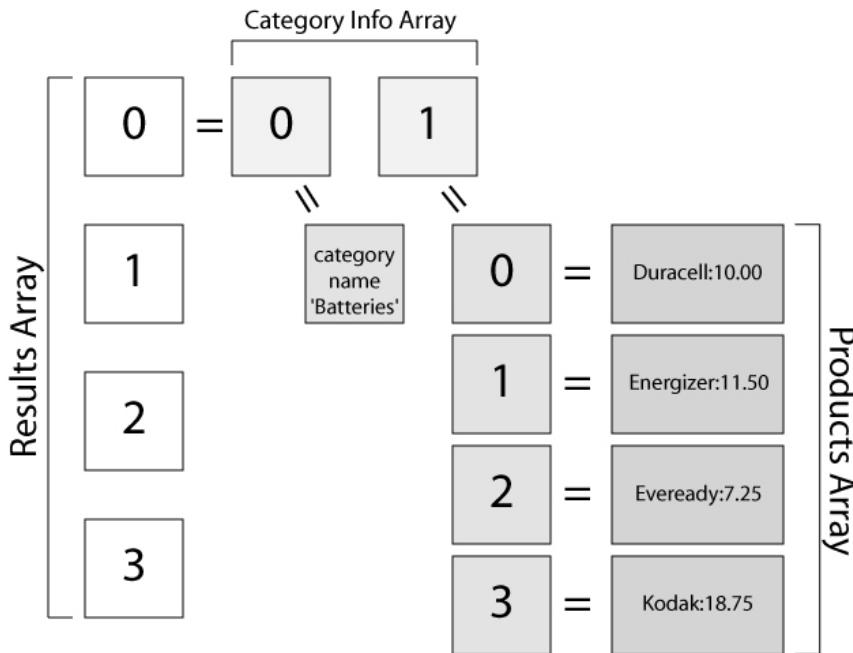


Figure 5. A conceptual 3D array for the Batteries category.

2. Open the **JMartSale.html** file located in this lab's directory. It is basically a simple page with a header, a form, and one button to launch the `showArray` function.
3. Open the **JMartSale.js** file. At the top of the document, under the comments, create a variable called `categories` and define it with a comma delimited string: “**Batteries,Soap,Magazines,Tires**”. Create an array with four positions, using the statement `var aProducts = new Array(4);`. Define the products as double-delimited strings; each product is delimited with a comma, and within each comma the product is delimited from its price with a colon. Use the values from Table 1.

Array Index	Value
aProducts[0]	“Duracell:10.00,Energizer:11.50,Eveready:7.25,Kodak:18.75”
aProducts[1]	“Dial:1.50,Dove:.99,Irish Spring:112.14”
aProducts[2]	“Sports Illustrated:3.95,Mac Addict:5.25,Maxim:9.99,Us Weekly:5.21,Time:5.01”
aProducts[3]	“Pirelli:120.00,Goodyear:89.32,BF Goodrich:92.52,Firestone (Ford blowout sale):0.00”

Table 1. Values for aProducts array.

```
var aProducts = new Array(4);
aProducts[0] = "Duracell:10.00,Energizer:11.50,
                Eveready:7.25,Kodak:18.75";
aProducts[1] = "Dial:1.50,Dove:.99,
                Irish Spring:112.14";
aProducts[2] = "Sports Illustrated:3.95,
                Mac Addict:5.25,Maxim:9.99,
                Us Weekly:5.21,Time:5.01";
aProducts[3] = "Pirelli:120.00,Goodyear:89.32,
                BF Goodrich:92.52,Firestone
                (Ford blowout sale!):0.00"
```

4. Under **aProducts**, create a new function called **createArray()** that accepts **catlist**, **products**, and **delim**. The function will expect a comma delimited string, an array of comma/colon delimited strings, and a delimiter (as a string). Create another function below it called **showArray()**, that takes no parameters.

```
function createArray(catlist, products, delim) {
}

function showArray() {
```

5. In **createArray()**, you are going to create a 3D array out of the data provided. First, split **catlist** into an array called **cats**, using the statement **var cats = catlist.split(delim);**. Remember **string.split(delim)** creates an array that contains the divided string elements. Next, define the array that will be returned from this function, using the statement **var aresult = new Array();**.

```
function createArray(catlist, products, delim) {
    var cats = catlist.split(delim);
    var aresult = new Array();
```

6. On the next line, create a for loop with the condition (**ca = 0; ca < cats.length; ca++**), which will iterate once for each category in the **cats** array. Within the loop, create a temporary array to hold the product elements within that category, using the statement **var tmparray = new**

**Array(2);** This array only needs to be a size two because it will hold the category name in the zero index position, and an array of size two that contains the product and price in the second index position.

```
for (ca = 0; ca < cats.length; ca++) {  
    var tmparray = new Array(2);  
}
```

7. Next, add the current category in the cats array to the temporary array, using the statement **tmparray[0] = cats[ca];**. The categories are in the correct order to match up with the products located in **aProducts**. All you have to do is grab the list of products that match the current iteration of the loop. First, split the products into tmparray[1], using the statement **tmparray[1] = products[ca].split(delim);**.

```
for (ca = 0; ca < cats.length; ca++) {  
    var tmparray = new Array(2);  
    tmparray[0] = cats[ca];  
    tmparray[1] = products[ca].split(delim);  
}
```

8. Next, sort the products using the sort() method of **tmparray[1]**.

```
for (ca = 0; ca < cats.length; ca++) {  
    var tmparray = new Array(2);  
    tmparray[0] = cats[ca];  
    tmparray[1] = products[ca].split(delim);  
    tmparray[1] = tmparray[1].sort();  
}
```

9. Finally, add the **tmparray** to the **aresult** array, using the statement **aresult[ca] = tmparray;**

```
for (ca = 0; ca < cats.length; ca++) {  
    var tmparray = new Array(2);  
    tmparray[0] = cats[ca];  
    tmparray[1] = products[ca].split(delim);  
    tmparray[1] = tmparray[1].sort();  
    aresult[ca] = tmparray;  
}
```

10. Now you will create a quick loop to report the size of the **aresult** array, the size of each of its indices, and within each index, the size of the products array. Create the variable **msg** to display the results. Start **msg** by adding “Result array length: ” + **aresult.length** + “\n\n”;

```
var msg = "Result array length: " + aresult.length +  
"\n\n";
```

11. Next, create a for loop to add the array information to **msg**. It needs to iterate through the length of **aresult**.

```
for (ra = 0; ra < aresult.length; ra++) {  
}
```

12. Output the current index of **aresult** and the length of the category/product array in **aresult**’s current index (should always equal 2).

```
for (ra = 0; ra < aresult.length; ra++) {  
    msg += "Array at " + ra + "'s length is: " +  
        aresult[ra].length + "\n";  
}
```

13. Next, output the length of the product array located in index[1] of the category/products array. To reference the index of an array within an array, you can use the shortcut syntax format **arrayname[parentindex][childindex].methodname**.

```
        for (ra = 0; ra < aresult.length; ra++) {  
            msg += "Array at " + ra + "'s length is: " +  
                aresult[ra].length + "\n";  
            msg += " - Its product array length is: " +  
                aresult[ra][1].length + "\n";  
        }
```

14. After the loop, display the **msg** variable in an alert dialog box, and then return **aresult**.

```
for (ra = 0; ra < aresult.length; ra++) {  
    msg += "Array at " + ra + "'s length is: " +  
        aresult[ra].length + "\n";  
    msg += " - Its product array length is: " +  
        aresult[ra][1].length + "\n";  
}  
alert(msg);  
return aresult;  
}
```

15. In the **showArray()** function, add the statement **var marray = createArray(categories, aProducts, ",")**; to execute the **createArray()** method.

```
function showArray() {  
    var marray = createArray(categories, aProducts, ",");  
}
```

16. Test the exercise by launching **JMartSale.html** in your browser. You should see an alert dialog box with the specifics of the array created from the delimited strings, as shown in Figure 6.

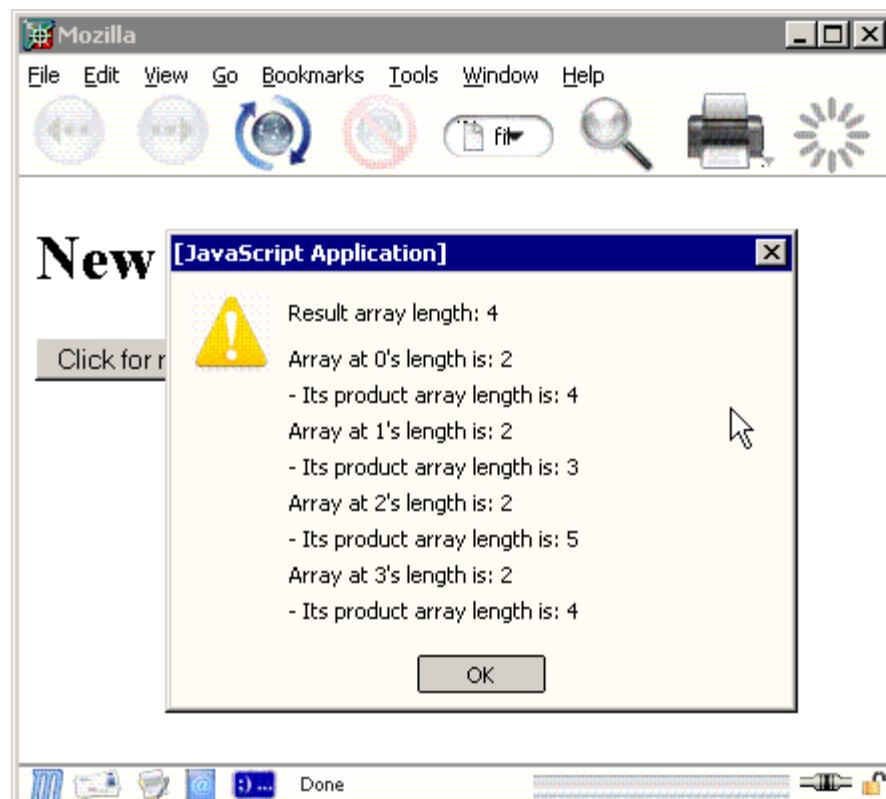


Figure 6. Array information generated by this exercise.

# Display the Product

## Objective

Learn how to display the data of the 3D array to your users. Commonly you would split the data up into a table where you could see the price and all the details. For this exercise you will create a more complicated version of the alert(msg) from the previous exercise using all the data.

## Things to Consider

- A comma is not the only delimiter you can use. In fact, you can use just about any character. A common delimiter is the pipe character '|'.
- If you had many lists of categories and products, you could use `array.join()` to add them all together and throw them into the `createArray` function.
- An array is JavaScript's most efficient way to keep collections of the data in a page. They are quite handy when a large amount of data is being handed to you from the server side, and are actually compatible with a Java ArrayList.

## Step-by-Step Instructions

1. Return to the `showArray()` function. Right below the line where the variable `marray` is defined, declare the products message, using the statement `var msg = "You know the prices are good\nwhen they are J-Prices!\n\n";`.

```
function showArray() {  
    var marray = createArray(categories, aProducts, ",");  
    var msg = "You know the prices are good\nwhen they are  
J-Prices!\n\n";  
}
```

2. Create a loop to iterate through `marray`. The value of `marray` at the current index is another array that will contain the category in its `index[0]` position and an array of products in its `index[1]` position. Get the category and add it to the `msg` string by first accessing `marray` with the statement `marray[ma]`, and then accessing `index[0]` of that value: `marray[ma][0]`.

```
    for (ma = 0; ma < marray.length; ma++) {  
        msg += marray[ma][0] + ":\n";  
  
    }
```

3. Now, under the category in **msg**, the application must list out the products located in index[1] of the current value of **marray**. Create a for loop to iterate through this sub array, using the length of the products array in the counter by calling **marray[ma][1].length**.

```
    for (ma = 0; ma < marray.length; ma++) {  
        msg += marray[ma][0] + ":\n";  
  
        for (ia = 0; ia < marray[ma][1].length; ia++) {  
  
    }
```

4. Now create an even smaller array that contains the name and price of the current product in the for loop. To access the string that contains the name and price, you must first access the outermost array, using the notation **marray[ma]**, then access the secondary products array with the notation **marray[ma][1]**, and finally access the current product in the products array using the notation **marray[ma][1][ia]**. From there, you can use the **split()** method to split the string values at the colon.

```
    for (ma = 0; ma < marray.length; ma++) {  
        msg += marray[ma][0] + ":\n";  
  
        for (ia = 0; ia < marray[ma][1].length; ia++) {  
            var product = marray[ma][1][ia].split(":");  
        }
```

5. If you are getting confused, review the conceptual array for the Batteries category in Figure 7.

## Lab 4: Arrays

---

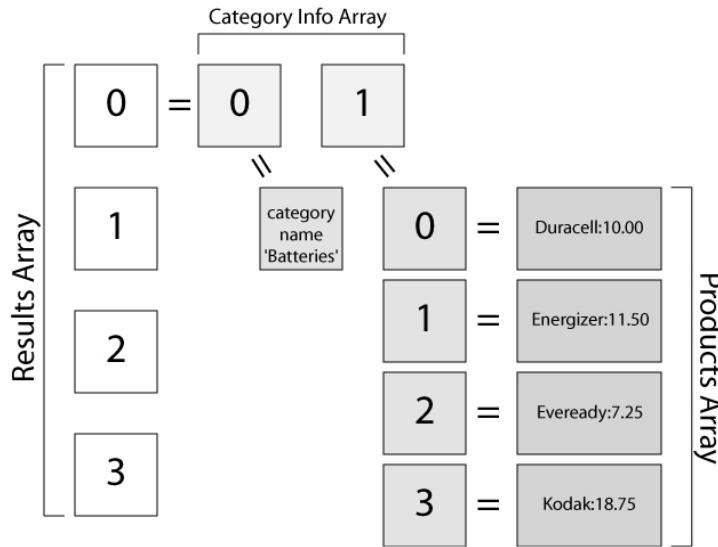


Figure 7. Conceptual array review.

6. For the next line, create an output message to describe the product and its price. First, display the product's number in the message using the statement **msg += (ia + 1)**, then append it with a separator + “ – ” +, and then follow that with the product name + **product[0]**.

```
for (ia = 0; ia < marray[m][1].length; ia++) {  
    var product = marray[m][1][ia].split(":");  
    msg += (ia + 1) + ". - " + product[0];
```

7. Next, add the string “ **at a low, low price of: \$**”, followed by the price, **product[1]**. Finally, add a line break, using the characters + “\n”;

```
for (ia = 0; ia < marray[m][1].length; ia++) {  
    var product = marray[m][1][ia].split(":");  
    msg += (ia + 1) + ". - " + product[0];  
    msg += " at a low, low price of: $ "+  
          product[1] + "\n";  
}
```

8. In the code for the main for loop, add the statement **msg += “\n”;** on the line directly after the inner for loop. This adds a return in between categories:

```
for (ma = 0; ma < marray.length; ma++) {  
    msg += marray[ma][0] + ":\n";  
  
    for (ia = 0; ia < marray[ma][1].length; ia++) {  
        var product = marray[ma][1][ia].split(":");  
        msg += (ia + 1) + ". - " + product[0];  
        msg += " at a low, low price of: $" +  
            product[1] + "\n";  
    }  
  
    msg += "\n";  
}
```

9. Finally, outside of the original for loop, add the statement **alert(msg);**, which will display the products for the user.

```
msg += "\n";  
}  
  
alert(msg);  
}  
//-->
```

10. Test the exercise by launching **JMartSale.html**. Once the button is pressed, you should see the original alert from the first exercise with the **aresult** array information. Following that you will see the sales list from J-Mart, put together tidily in an alert box, as shown in Figure 8.

## Lab 4: Arrays

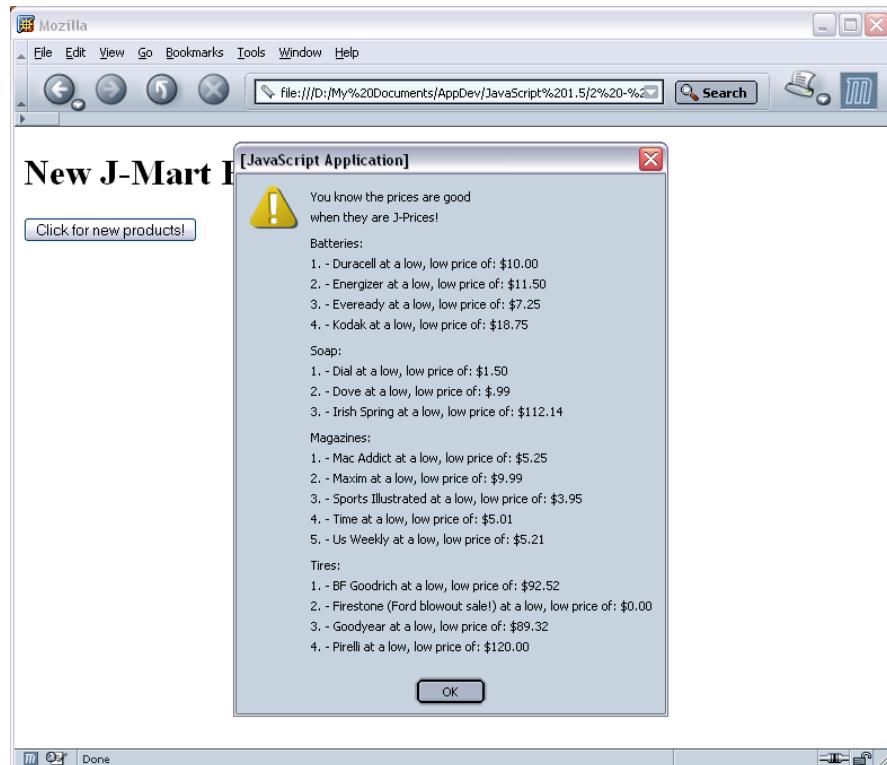


Figure 8. The Alert results for the J-Mart sale.

# Form Interaction

## Objectives

- See how to control the various form objects that are available to you.
- Learn to use complex controls like select-one objects, radio objects, and check box objects.
- Discover how to retrieve the values from a form and work with them in your code.
- Understand how event handlers can be used with the form and other objects.
- Learn about the two main validation techniques and how you can apply them.

# Working with Form Objects

Arguably, the most common use of JavaScript is to interact with XHTML forms. Most Web sites have forms for various purposes, ranging from a simple text box that you use to submit a query term to your favorite search engine, to a personal information form for joining a mailing list, to a complex server-driven form for placing e-commerce orders online. The easiest, most readily available way to get any type of input from an XHTML page is with a form.

## Alternatives to Forms

Before exploring the form object in detail, you should be aware that forms are not the only way to retrieve data from a user. Plug-ins such as Java applets, Flash movies, Shockwave movies, and Active X controls are some other popular ways for users to enter data in different ways. For example, Sferyx.com offers an HTML GUI editor written as an applet, as shown in Figure 1.

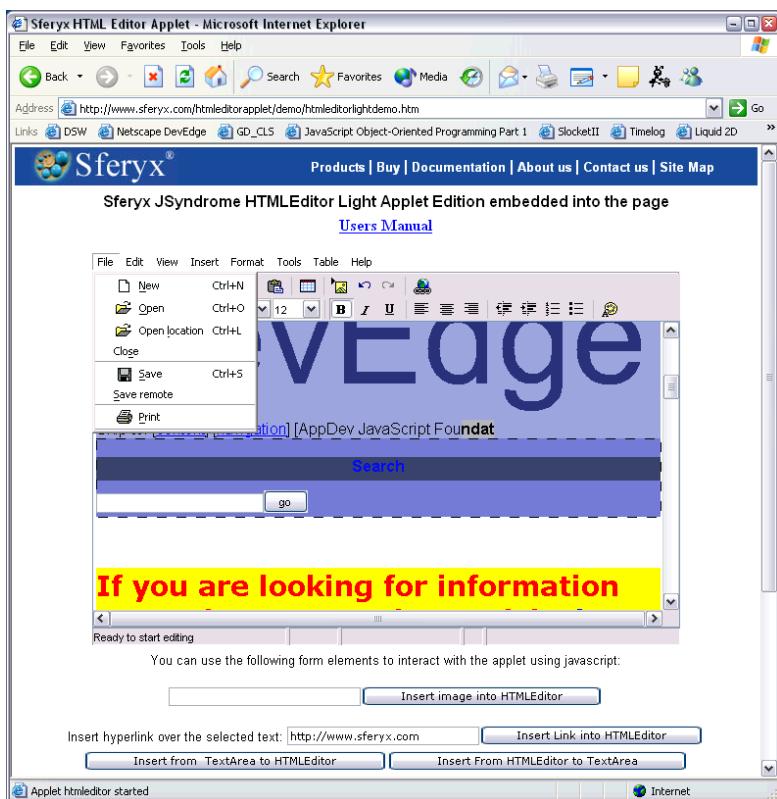


Figure 1. Sferyx.com's HTML GUI editor (Java applet).

Figure 2 shows a page from the Web site Shutterfly.com, which offers Internet Explorer users the ability to upload multiple photographs at once via an Active X control.

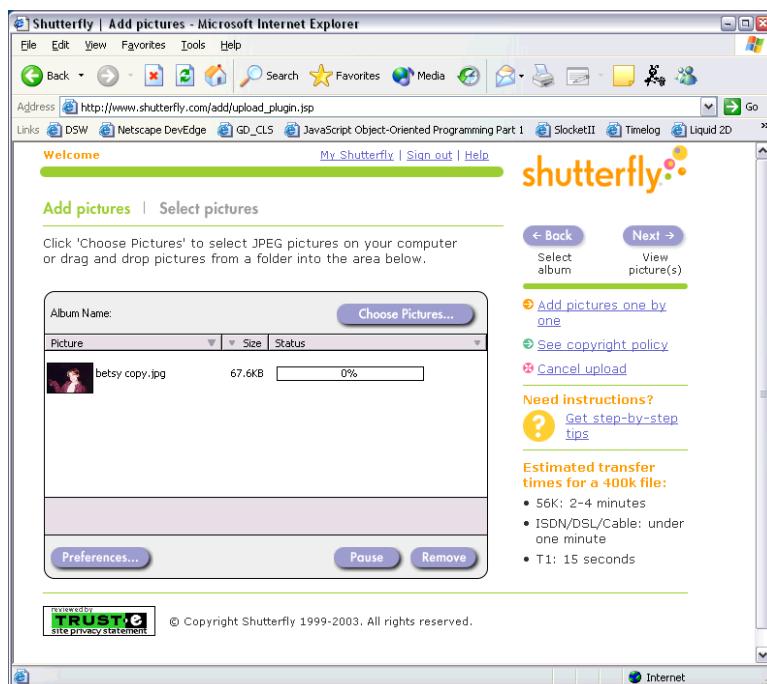


Figure 2. Shutterfly Active X control for picture uploads.

You have probably encountered a Java applet that had button controls, and may have noticed that these buttons differ in appearance from typical XHTML form buttons. They differ because they are rendered with Java's AWT or Swing graphics. Macromedia's Flash and Shockwave movies can also represent controls in just about any possible way, because they are completely graphics-based. It is worth noting that Flash movies are scripted using Action Script, which is a JavaScript-like scripting language.

Plug-ins run within the context of the browser, and the programming behind them enables the developer to create their own environment within the browser window. While these plug-in development environments offer a lot of creative latitude, they also require developers to learn the vendors' proprietary languages and tools to develop them. Further, the users must have installed specific plug-ins on their computers before the plug-in code can run.

When it comes to data entry, perhaps the most straightforward way to handle users' input is with forms. JavaScript is supported by default in almost all modern browsers, and does not require that you purchase any proprietary tools to write the code. If you are an XHTML master, you also know that forms can be even more attractive than custom plug-ins.

## Form Basics

Because of the many different approaches to take when you work with forms, it is advisable to find a good JavaScript reference that details the properties and methods of the various form objects. Table 1 lists the main properties of the form object, as a reference.

Property	Description
acceptCharset	List of one or more character sets that the server receiving this form will recognize.
action	URL on the server that the form will submit data to.
autocomplete	Set to on or off. This is an Internet Explorer feature that allows the browser to supply hints based on previous inputs into forms.
elements	An array of XHTML elements within the form.
encoding	Define the MIME type of data going to the server. For example, a mailto: url would use 'text/plain'.
enctype	This is the W3C dom model. NN6 provides both for backwards compatibility.
length	Same as the length property of the elements array—returns the amount of elements in the form.
method	GET or POST. POST data is wrapped into the http header and is more secure, GET data is mapped into the URL via a querystring: <a href="http://www.google.com/search?q=javascript">http://www.google.com/search?q=javascript</a> . This sets the value of 'q' to 'javascript' and submits the data to Google's search.
name	JavaScript reference name of the form within the page.
target	When submitted, the results are usually sent back to the current page. If you are working in a framed environment, you may want the results sent back to a different frame. This is where you would use the target attribute.

Table 1. Form specific properties.

Table 2 lists the main methods of the form object.

Method	Description
handleEvent()	Accepts an event. This is used when the form needs to capture and be ready to handle a particular event.
reset()	Clears all the current values in this form's elements.
submit()	Submits the form.

Table 2. Form specific methods.

Table 3 lists the event handlers that are available for the form object.

Event Handler	Description
onReset	When the Reset button is clicked in this event handler, JavaScript will be read before the form is actually reset.
onSubmit	When the Submit button is clicked in this event handler, JavaScript will be read before the form is actually submitted.

Table 3. Form specific event handlers.

Some of the actions listed in Table 1 are used only occasionally in normal JavaScript code. The most commonly used aspects of form objects are the elements array property, the submit() method in Table 2, and the onSubmit event handler in Table 3.

It is important to understand a form's place in the document. An XHTML page can contain multiple form objects. In the DOM, references to these form objects are stored in a forms array (called forms) as a property of the XHTML page itself. Similarly, form elements are stored in an array called elements as a property of the form object. This means that both the forms and the elements of a specific form can be handled like a typical array. To access a form, you can use its name to specify it directly:

```
document.[formname]
```

Alternately, you can use the form's array index reference to access it indirectly:

```
document.forms[int]. // int is the form's array index
```

See *FormDump.js*  
(use with *TestFormDump.html*)

The following code snippet lists all the main properties of all the forms, and the types and values of the form elements, in an XHTML document. This can be useful when testing the code on a page you are troubleshooting.

```
<!--
//Quick Form Dump
var s = "";
for (df = 0; df < document.forms.length; df++) {
    var f = document.forms[df];
    var b = "<br/>";
    var sp = "&nbsp;";
    s += "<h2>" + f.name + ", or forms[" +
    s += df + "]</h2>";
    s += sp+sp + "<b>action</b> = " + f.action + b;
    s += sp+sp + "<b>elements[].length</b> = ";
    e = f.elements; //assign the elements array to e
    s += e.length + b + b;
    s += "<table cellspacing=\"4\">";
    s += "<tr bgcolor=\"#efefef\"><th>name</th>";
    s += "<th>type</th><th>value</th></tr>";
    for (el = 0; el < e.length; el++) {
        s += "<tr>";
        s += "<td>" + e[el].name + "</td>";
        s += "<td>" + e[el].type + "</td>";
        s += "<td>" + e[el].value + "</td>";
        s += "</tr>";
    }
    s += "</table>";
    s += (b + b);
}

document.write(s);
document.close();
//-->
```

## Form.method Property

The method property of a form defines how the data is sent to the action location. You can set the method either explicitly as GET or POST, or it is set to GET by default if you do not specify a method property. The GET method inserts form data into the query string, which is appended to the URL specified in the action property of the form. On the target page, the URL will look something like this:

```
http://www.url.com/cgi?name=bob&submit=Submit+Form
```

The string that starts at the question mark and goes to the end of the line is referred to as the query string. You can access and manipulate the values passed via the GET query string on your target page using JavaScript's location.search method, which is covered later in this chapter.

**NOTE** You can pass a maximum of 255 characters as part of a URL. Check to be sure that your URL does not exceed this limit if you are using the GET method, to prevent your data from being truncated.

The POST method is used to send greater quantities of data than the query string can handle, and send it more securely. POST data is not shown in the query string; but instead is encrypted into the http request object that is sent to the server. Unfortunately, JavaScript cannot retrieve the values submitted via the POST method.

## Form.action Property

The form action defines what happens to the form data once a submit is executed. Typically, the action property will be a URL to a CGI script, Java servlet, or some other type of application. You also have control over setting the action. Suppose you have three CGI applications waiting to handle different data, and depending on what the user chose or entered, the form needs to direct the data to one of these applications. In your script, you can catch the form data, determine the action that needs to occur, and send the data on to the correct application. You access a form's action as follows:

```
document.[formname].action =  
"http://www.url.com/cgiscript";  
var actionString = document.formname.action;
```

## Fieldsets

You can organize blocks of form elements with the XHTML fieldset tag. The fieldset creates a physical border around the elements and provides a title area or *legend* to the group, as shown in Figure 3.

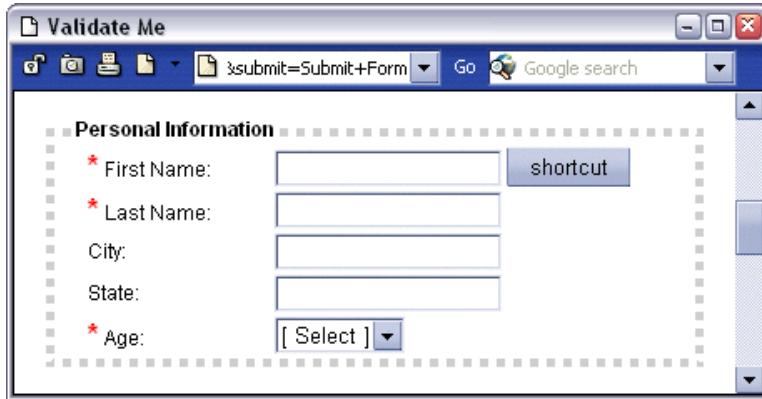


Figure 3. A fieldset example.

As you can see in the following code, the fieldset displayed in Figure 3 is constructed with the legend *Personal Information* and is assigned a style with a 5-pixel dotted gray border:

```
<style type="text/css">
<!--
    .fieldset {border: 5px dotted #CFCFCF; }
-->
</style>
<fieldset class="fieldset">
<legend>Personal Information</legend>
    <div align="center">
        Hi, I'm in a fieldset!<br/>
        This is personal information!
    </div>
</fieldset>
```

TIP: You can also create “mouse-over” text in a form element by assigning a title property to it: <input type=”text” name=”firstname” title=”First Name”/>

The fieldset is very useful for organizing radio groups, much like you would see in a Windows application. If you want to learn more about cascading style sheets, pick up a reference book on CSS2.

# Form.Elements[]

As you saw in the Quick Form Dump script, the `form.elements[]` property provides you with an array to access all the elements of the form. This allows you to iterate through the form elements, and select specific elements and handle them individually.

A good example of the `form.elements[]` array at work appears on Microsoft's Hotmail site. In the e-mail list view, the header contains a check box for selecting all of the e-mails for a mass move or delete operation.

You can create this Select All feature by iterating through the `form.elements` array, using the `elements.length` property to dynamically determine the array's size. Your code should identify each check box element that corresponds to e-mail messages, then update each check box's value to true to select it. When the check box object's value changes, it will automatically update the on-screen check box image to a checked state. Upon submission, the form will be processed by a server-side application to handle deleting or moving the selected e-mails.

*See Form  
Elements.html*

The following code example demonstrates the power of the `elements` property:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
<script language="JavaScript">
<!--
// A Simple iteration of the give form's element's values.
function aVals(thefrm) {
    var msg = "Form " + thefrm.name + "'s Values\n\n";

    for (fe = 0; fe < thefrm.elements.length; fe++) {
        element = thefrm.elements[fe];
        msg += element.name + "=" + element.value + "\n";
    }

    alert(msg);
}
//-->
```

```

</script>
</head>
<body>
    <form name="f1">
        <input type="hidden" name="hiddentest"
               value="test"/>
        <input type="button" name="showbutton"
               value="Show Form"
               onClick="aVals(document.f1);"/>
    </form>
</body></html>

```

The result is an alert box that displays the element's values, as shown in Figure 4.



Figure 4. An Opera 7 Alert box displaying form f1's values.

## Text Objects

You will interact frequently with several types of text objects in JavaScript, examples of which appear in Figure 5. For example, the previous code snippet contained a hidden object that had a value property like a text object, but whose purpose is simply to hold data behind the scenes instead of displaying it on the screen. You will also encounter the password's text object, which is essentially a text box that displays only asterisks for each letter of input, to hide the value that the user is typing. In addition, you'll also use an object called a textarea, which can display multiple lines of text.

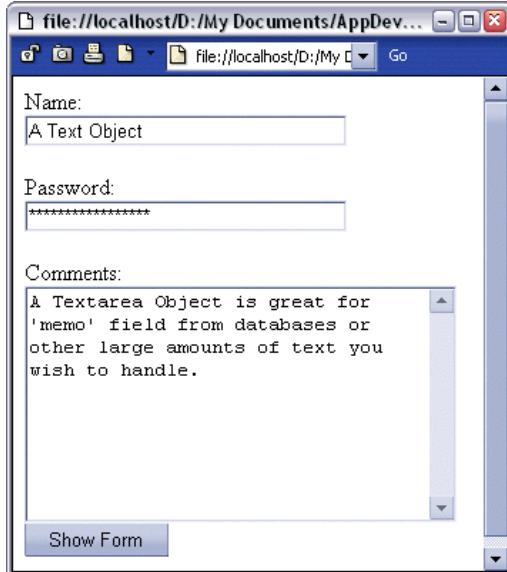


Figure 5. Textbox, Password, & TextArea objects.

The following three tables (Table 4, Table 5, and Table 6) lay out some of the properties, methods, and event handlers for text type objects.

Property	Description
defaultValue	Set this so it grabs the original value of the object after it has been changed.
form	If you happen to pass an element to a script, you can use its form property to find out which form the element is located in.
maxlength	Maximum allowable characters for an element's value.
name	The name of this element. Used in JavaScript to reference the element, and also the name that is passed when the form is submitted.
readOnly	Makes the element read-only.
size	The size, in characters of the current value of the element.
type	Type returns the kind of object the element is, for example, select-one.
value	The current value of this element.

Table 4. Text type object properties.

Method	Description
select()	Call this method of an element when you want to select that element.
focus()	Call this method to gain focus on the specified element.
blur()	Call this to leave focus on an element.

Table 5. Text type object methods.

Event Handler	Description
onChange	This event handler fires when the value of an element changes.
onSelect	When an element is selected, this event handler fires.
onFocus	When the element gets focus, this event handler fires.
onBlur	When you click elsewhere on the page, or tab off of the textfield, this event handler fires.
onSelect	When the contents of the textfield are selected, this event handler fires.
onClick	When the mouse is clicked on the textfield this event handler fires.

Table 6. Text type object event handlers.

Manipulate text objects in JavaScript code with the same type of dot syntax that was used in earlier examples to access the form elements:

```
document.[formname].[textfieldname].value
//or type, size, etc...
```

The values associated with text objects or form objects are only available as long as the page that contains them exists. Once you submit or refresh that page, the form data will be lost unless you explicitly forward it via a GET or POST method. Depending on how the browser caches the data, the user may be able to retrieve the values by scanning through the browser's history with the back and forward buttons.

You should be aware of some other lesser-known properties and methods. First, on some occasions you might not want a text object to be editable or even enabled. You can accomplish this in three ways. For the version 4 and version 5 browsers you can use the text object's **READONLY** or **DISABLED** properties to accomplish this objective. In addition, you can add an **onFocus** event handler to the object that executes the **.blur()** method, which will prevent the user from accessing the object:

```
<input type="text" name="text" DISABLED/>
<input type="text" name="text1" READONLY/>
<input type="text" name="text2" onFocus="this.blur()"/>
```

**NOTE** Before Internet Explorer 4 and Netscape 6 you could not disable text objects, so the event handler method was the only alternative.

Event handlers are useful with text objects when you want to store data input from a Web application. If the potential for entering invalid data exists, you may want to be able to save the original value, just in case you want to reference it again.

If you want to select the contents of a text object in JavaScript, you can use its select() method. This does not set focus on the object in the browser, but instead only selects the object's contents. To set focus on the object in the browser window, you can call the object's focus() method. Likewise, to disable the text object in the browser, you would call its blur() method.

The following code snippet illustrates how these methods and event handlers can be used in a script:

```
<script language="JavaScript">
    var oldvalue;
    function storeVal(value) {
        oldvalue = value;
    }
</script>

</head><body>
    <input type="text" name="name"
          value="storeme!"
          onFocus="storeVal(this.value); this.select();"
          onChange="this.value=oldvalue; this.focus();"/>
    ...
</body>
```

In this case, the user can edit the text object, but the original value returns the moment the user tabs off of the text object. When the user clicks on the text object, the onFocus event fires, calling the storeVal() function and passing it the previous value of the text field, which caches it in the variable called oldvalue. The previous contents are then conveniently selected so the user can enter new contents. When the user attempts to change the text object, the onChange event fires, restoring the previous value and setting the focus to the text object.

You should also be aware of some additional properties for the textarea object. To define the size of a textarea, you must set its cols and rows properties. The cols property refers to the width of the textarea in terms of the number of characters it can display, while the rows property refers to the number of rows it can display. Another property of the textarea is the wrap property, which determines whether strings are wrapped to the next row when they reach the column limit. Values can be hard (wraps the complete word), soft (wraps the next character), or off (no wrapping).

In a textarea it is also helpful to know how carriage returns work. UNIX, Macintosh, and Windows environments all handle carriage returns differently.

Table 7 lists the carriage return characters used in each operating system.

O.S.	Characters
Macintosh	\r
Unix	\n
Windows	\r\n

Table 7. Carriage Returns in the <textarea> tag.

## Button Objects

You can use the button object in several ways. First, you can use the XHTML button element tag, which is not directly related to the form, but can be extremely useful when creating JavaScript applications. You can also use the form-related buttons: submit, reset, and image. When the user clicks the submit button, it activates the form's submit event, or onSubmit. The reset button clears the values for any elements within the form, with the exception of the values of the buttons themselves.

```
<button value="submittedvalue">Click Me, I'm a  
button.</button>  
<form>  
<input type="button" name="buttonname" value="Click Me"  
    onClick="activateFunction();alert('or statement');"/>  
<input type="submit" value="Submit Form"/>  
<input type="reset" value="Clear Form"/>  
</form>
```

The values for each of these objects are passed to the form's action when it is submitted, regardless of whether it is used in any subsequent pages.

## **Image Inputs**

The image version of the submit button, img, is used when you want to display a graphical representation of a button rather than a standard submit button. Image inputs are handled in much the same way as submit buttons, with one added property: src. The src property defines the location of the image to use in place of the standard button, much as it does with the tag in XHTML. The difference between the two image types is that you have more control over an XHTML image.

## **Check Box Objects**

The check box is a simple control for handling Boolean values; either it is checked (true) or unchecked (false). You can access the checked property of the object to access or change the check box's current state. In addition, you might want to know whether the check box was checked by default, which you can determine via its defaultChecked property.

Like most form objects, check boxes also have a value property. Check box behavior is also a little different at submission time. If a check box is unchecked, its object and value are NOT passed to the form's action; the check box object is only passed along to the target if it is checked, thus reducing the form's "overhead."

```
<input type="checkbox" name="mentalstate"  
    value="Doin' Great."/>
```

In script, you put a check in the check box object by setting its checked property to true:

```
<!--
document.form.checkbox.checked = true;
//-->
```

**TIP:** When a check box or radio button is checked via script, the onClick event of the check box will not fire.

The click() method is used in the older browsers to mimic clicking a check box in script. While modern browsers maintain support for the click() method, it is much wiser to ignore this method and rely on the check box's checked property in your scripts instead. The rationale for this is that the click() method often displays errant behavior, and manipulating the checked property gives you more control over the check box object. These same issues are true for the click() method of the radio object as well.

## Radio Objects

Radio buttons are perfect for multiple-choice inputs where only one possible answer exists. Set up a radio group by creating multiple radio inputs that have the exact same name but different values. To help the user identify that these options are in a single group as shown in Figure 6, consider adding them to a fieldset as described in the beginning of this chapter.



Figure 6. A radio group in a fieldset.

Radio button properties are similar to those for the check box object. A radio input has a checked property, defaultChecked property, a click method, and an onClick method. One significant difference is that radio options are aggregated into a named group, and the group consists of an array of radio options. The radio group can be accessed using the following syntax:

```
document.[formname].[groupname]
```

You can determine the number of radio input elements within the group by using the group's length property. This is handy when you must iterate through a radio group to find out which radio input element is checked:

```
<!--
....form = document.formname;
....for (rl = 0; rl < form.radiogroup.length; rl++) {
.....if (form.radiogroup[rl].checked) {
.....alert("Check val is:");
.....alert(form.radiogroup[rl].value);
.....}
....}
-->
```

## Select and File Objects

Select objects are used to display multiple values in a selectable list and are an excellent way to save space in a user interface. A select object can take the form of a drop-down list or a list box, depending on the value assigned to its size property.

If the object's size is set to 1, only one item from the list will display at a time. The user can click on the button at the right to open the list of possible selections. If the object's size is greater than 1, then the select object will display as a scrollable list box. The list box will display a number of rows corresponding to the value of its size property, as Figure 7 shows.

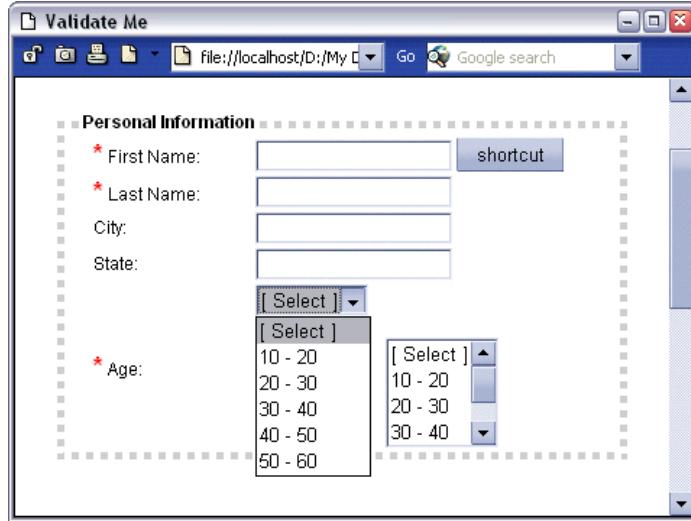


Figure 7. The difference in select pop-up and select list.

The select object is more complex than the other form objects. The select object is actually a combination of two objects: the parent select object itself, and a set of several option objects. Each option object within the select object represents a possible selection, and these are stored in the options array property of the select object. You can add and remove options on the select object dynamically via JavaScript code. Finally, when a select object has two or more options, you can specify whether the user can select multiple entries, or just one at a time.

To learn more about how select pop-ups work in JavaScript, you can try adding and removing options dynamically. First, you need to understand how they are laid out in XHTML:

```
<select name="ages" size="1" title="Ages">
    <option value="" SELECTED>[Select]</option>
    <option value="1">10 - 20</option>
    <option value="2">20 - 30</option>
    <option value="3">30 - 40</option>
    <option value="4">40 - 50</option>
</select>
```

To build this out dynamically, you can start with a simple array or build the labels (the text that appears between the option tags) programmatically. For the sake of simplicity, the examples use an array.

First, you will want to create an XHTML document with select objects. One should be a pop-up style, the other a list style:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    </head>
  <body>
    <form name="form">

      <!-- SELECT POPUP -->

      <select name="ages" size="1" title="Ages Popup">
        </select>

      <input type="button" value="Fill Select"
            onClick="fillSelect(document.form.ages)"/>
      <input type="button" value="Alert Selection"
            onClick="selectionAlert(document.form.ages)"/>
      <input type="button" value="Clear List"
            onClick="clearList(document.form.ages)"/>

      <br/><br/>

      <!-- SELECT LIST -->

      <select name="ageslist" size="3" title="List of Ages"
              MULTIPLE>
        </select>
```

```
<input type="button" value="Fill Select"
       onClick="fillSelect(document.form.ageslist)"/>
<input type="button" value="Alert Selection"
       onClick="selectionAlert(document.form.ageslist)"/>
<input type="button" value="Clear List"
       onClick="clearList(document.form.ageslist)"/>

</form>
</body>
</html>
```

As in most JavaScripts, it is best to organize the functionality into separate functions. The first function for select handling will just set up the array and call other utility functions that fill a select object, clear a select object, and alert the selected value(s) of a select object.

```
<script language="JavaScript">
<!--
function fillSelect(slct) {
    var agelist = new Array(6);
    agelist[0] = "10 - 20"; agelist[1] = "20 - 30";
    agelist[2] = "30 - 40"; agelist[3] = "40 - 50";
    agelist[4] = "50 - 60"; agelist[5] = "60 - 70";

    //Make sure select object is empty!
    clearList(slct);

    //Add our options;
    addList(agelist, slct);
}
```

Now, the clearList function needs to be set up. The best way to clear a select object is by using a while loop. You initiate it by handing off the length (the while object has length) and use the select remove method to delete each option object that is in the array:

```
function clearList(slct) {  
    while (slct.options.length) {  
        slct.remove(0);  
    }  
}
```

**TIP:** In older browsers, you might have to use the option objects remove method, instead of the select objects: slct.options.remove(0);

Populating the list is easy. A function, addList, needs to be added that accepts an array of options and the select object to fill:

```
function addList(arraylist, slct) {  
    //Adds default instruction  
    slct.options[0] = new Option("[select]");  
    slct.options[0].value = "";  
  
    for (al = 0; al < arraylist.length; al++) {  
        slct.options[al + 1] = new Option(arraylist[al]);  
        slct.options[al + 1].value = (al + 1);  
    }  
}
```

Finally, it is helpful to know how to access the values. The difference between a pop-up and list select object is very clear here. With the pop-up, only one selection is possible, so all that needs to be done is to identify the selectedIndex and the option value at that index position. You do this by using the select objects selectedIndex as the index for the option in the options array (options[selectedIndexValue]).

Things are a bit different with the select list. Since there can be many selectable objects in a list (if it is set to MULTIPLE) you must iterate through all the options in the options array and identify the selected ones.

**NOTE** To select multiple options in a list, click on a single option, then press **CTRL** while you click on other options. You can also click on a starting option, and **SHIFT**-click further down in the list to select all the options between.

*See Select  
Example.html*

The following code displays the entire application. It creates two select objects: one in the single line style, and other in a multiple line style with multiple selections enabled. Buttons are added to alert the selection or selections you choose, and other buttons are created that fill and clear the available selections for each select object.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
<head>
<script language="JavaScript">
<!--
function fillSelect(slct) {
    var agelist = new Array(6);
    agelist[0] = "10 - 20"; agelist[1] = "20 - 30";
    agelist[2] = "30 - 40"; agelist[3] = "40 - 50";
    agelist[4] = "50 - 60"; agelist[5] = "60 - 70";

    //Make sure select object is empty!
    clearList(slct);

    //Add our options;
    addList(agelist, slct);
}

function selectionAlert(slct) {

    //Check for options.
    if (slct.options.length > 0) {
        if (slct.size <= 1) {
            //Popup detected.
            if (slct.options[slct.selectedIndex].value) {
                var msg;
                msg = "select: " + slct.name;
                msg += "\ntext: " +
                    slct.options[slct.selectedIndex].text;
            }
        }
    }
}
```

```
msg += "\nvalue: " +
       slct.options[slct.selectedIndex].value;
alert(msg);
} else {
    alert("Select an option first!");
}
} else {
//List detected.
var msg = "Select an option first!";
var areselections = false;
for (lo = 0; lo < slct.options.length; lo++) {
    if (slct.options[lo].selected == true) {
        if (!areselections) {
            msg = "Selected options for " +
                  slct.name + "\n\n";
            areselections = true;
        }
        msg += slct.options[lo].text;
        msg += " = " +
               slct.options[lo].value + "\n";
    }
}
alert(msg);
}
} else {
    alert("No Options!");
}
}

function addList(arraylist, slct) {
    slct.options[0] = new Option("[select]");
    slct.options[0].value = "";

    for (al = 0; al < arraylist.length; al++) {
        slct.options[al + 1] = new Option(arraylist[al]);
        slct.options[al + 1].value = (al + 1);
    }
}
```

```
}

function clearList(slct) {
    while (slct.options.length) {
        slct.remove(0);
    }
}

//-->
</script>
</head>
<body>
<form name="form">

<!-- SELECT POPUP -->

<select name="ages" size="1" title="Ages Popup">
</select>

<input type="button" value="Fill Select"
       onClick="fillSelect(document.form.ages)"/>
<input type="button" value="Alert Selection"
       onClick="selectionAlert(document.form.ages)"/>
<input type="button" value="Clear List"
       onClick="clearList(document.form.ages)"/>

<br/><br/>

<!-- SELECT LIST -->

<select name="ageslist" size="3" title="List of Ages"
        MULTIPLE>
</select>

<input type="button" value="Fill Select"
       onClick="fillSelect(document.form.ageslist)"/>
<input type="button" value="Alert Selection"
       onClick="selectionAlert(document.form.ageslist)"/>
<input type="button" value="Clear List"
```

```
onClick="clearList(document.form.ageslist)"/>

</form>
</body>
</html>
```

Congratulations! You are now a master of the select objects. Another interesting selection-based object is the file object.

## File Input Object

*See*

*FileInputExample.html*

The file object is a very simple one: in a document, the file object renders as a text object plus a browse button. The value becomes whatever file is selected. One specific property should be set when your code allows the user to upload a file to the server: the enctype property of the form must be set to multipart/form-data. This encryption type enables a file transfer to the CGI application.

**TIP:** You must have a CGI application that is capable of accepting files from Web pages for this object to be of much use.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head></head>
<body>
<form method="post" name="form1" action="fileCGI"
      enctype="multipart/form-data">
```

Please select a file:

```
<input type="file" name="filelocation" size="30"/>
<br/><br/>
```

```
<input type="button" value="Alert File Location"
       onClick="alert(document.form1.filelocation.value);"/>

</form>
</body>
</html>
```

# Validation and the onSubmit Event

Once the form is built and you have gone to great lengths to create dynamic select objects, well-positioned fieldsets, and form objects, you might want to make sure the user can't mess up all your hard work by entering the data improperly.

There are several ways to catch invalid values in a submission. The first way is to create a false submit button using a standard button object (as opposed to a form's actual submit button). When the user clicks this button, you can use its onClick event to call functions that evaluate each individual object type, if it is required, and then submit the form.

Another way to perform input validation is to catch a submission after a submit object is selected using the form object's onSubmit event. The onSubmit event can be programmed to call a validation function, which returns true if the data is valid and false if it is not. The form must wait to receive a return value of true from the function before it submits the form. If a value of false is returned, the form will not be submitted.

See  
**onSubmitExample.html**

In this example there are actually two forms and two functions to handle the forms. The form has an onSubmit event on it; it will fire if a script calls formname.submit(). Here is a sample XHTML document to try it out:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
  "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
  <head>
    <script language="JavaScript" type="text/javascript">
    </script>
  </head>
  <body>
    <form name="form1" onSubmit="return regularSubmit(this);">
      <input type="checkbox" name="checkSubmit"
        value="submitme" />
      Submit w/ submit button??<br/><br/>
      <input type="submit" name="submitbutton" />
    </form>
  </body>
</html>
```

```
        value="Actual Submit"/>
    </form>
<form name="form2">
    <input type="checkbox" name="checkSubmit"
           value="submitme" />
    Submit w/ regular button?<br/><br/>

    <input type="button" name="fakesubmitbutton"
           value="Fake Submit Button"
           onClick="fakeSubmit(document.form2);"/>
</form>
</body>
</html>
```

Now you need the script:

```
<!--
if (location.search) {
    alert("Form values:\n\n" + location.search);
}

function regularSubmit(form) {
    if (form.checkSubmit.checked) {
        alert("Click OK to submit");
        return true;
    } else {
        alert("Sorry, no submission!");
        return false;
    }
}
```

```
function fakeSubmit(form) {  
    if (form.checkSubmit.checked) {  
        alert("Click OK to submit");  
        form.submit();  
    } else {  
        alert("Sorry, no submission!");  
    }  
}  
//-->
```

It all adds up about the same. If the browser does not support JavaScript, neither function will run, so you will have to rely on server-side validation. The most commonly used method of the two is the onSubmit route.

# Summary

- Forms are the most common way to retrieve data from clients, although many other ways exist including flash movies, Java applets, Active X controls, and more.
- Forms have many properties, methods, and event handlers to help you manipulate and filter data.
- Using the form method submit() programmatically submits a form once the conditions you define are achieved.
- A form's onSubmit event handler is helpful in making sure any last minute conditions are met before actually submitting the form.
- To easily access multiple forms in an XHTML document, you would use the forms.elements array, which has a reference to each form in the document via the form's index in the array: document.forms[0].submit().
- Out of the two form methods available, GET & POST, POST is the most secure method because data is contained in the HTTP request. With GET, the data is attached to a query string, which is visible in the address bar of a browser.
- By setting a forms action parameter, you can dynamically assign the address that the form will submit to.
- To obtain typed data from a user, you would use the text objects: The regular *text* object for non secure data, the *password* object for passwords and other secure data, and the *textarea* for large amounts of text.
- Form buttons are used for launching methods or any other custom actions. The submit button is a different object that submits the form. In addition, the reset button is used to clear any input into the form that contains it.
- Check boxes are helpful for true/false type values. If unchecked, the name/value is not sent when the form is submitted.
- Use radio buttons when a user must select one out of a certain number of options.
- When a selection out of a long list of items is necessary, you would use a select-one object, populated with objects that represent the items you want the user to select from.
- The file input type is used to retrieve the file location from the user's operating system as a string. Once submitted, a cgi must be programmed to accept the actual binary data of that file.

# Questions

1. What common form object would be perfect to add content from a database memo field?
2. True/False: Form objects can be read from JavaScript but not modified by JavaScript.
3. The collection that you want to iterate through to receive information about form objects is the `._____[]` array.
4. Validate a check box by using which property?
5. The dot syntax for accessing the options in a select object is `document.form.select._____[]`.
6. `form1` has a radio group of five radio input objects named `favSong`. How do you find out in script how many radio objects that group has?
7. When the user presses the submit button, what event handler on the form object catches it?
8. When you catch a form with that event handler, what must you precede a function call with?
9. Once validation in your script is complete, what do you return to the form from the original function that was called to submit the form?

# Answers

1. What common form element would be perfect to add content from a database memo field?  
**The <textarea> object.**
2. True/False: Form elements can be read from JavaScript but not modified by JavaScript.  
**False**
3. The collection that you want to iterate through to receive information about form objects is the .\_\_\_\_\_[] array.  
**elements**
4. Validate a check box by using which property?  
**checked**
5. The dot syntax for accessing the options in a select object is document.form.select.\_\_\_\_\_[].  
**options**
6. Assume that form1 has a radio group of five radio input objects named favSong. In script, what object property can be used to determine how many radio objects that group has?  
**document.form1.favSong.length**
7. When the submit button is pressed, what event handler on the form object catches it?  
**onSubmit=""**
8. When you catch a form with that event handler, what must you precede a function call with?  
**'return' example: onSubmit="return validateForm();"**
9. Once validation in your script is complete, what do you return back to the form from the original function that was called to submit the form?  
**true**

# Lab 5: Form Interaction

**TIP:** Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You will find all the completed code in **ValidateMe.html** and **ValidateMe.js**, in the same directory as the sample project. To avoid typing the code, use the exercise specific .js files **ValidateMe\_ex1.js**, for exercise one, and **ValidateMe\_ex2.js** and **ValidateMe\_ex3.js** respectively. Once you have completed each part, rename it **ValidateMe.js** to test it.

# Lab 5 Overview

In this lab you will learn to properly access various form elements, and how to validate values.

To complete this lab, you will need to work through three exercises:

- Terminal: Routing and Setup
- Defining Validations
- Display Results

Each exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

# Terminal: Routing and Setup

## Objective

In this exercise, you will use a decision construct to direct different form elements to the appropriate validation function.

## Things to Consider

- The type property for a select object returns as ‘select-one’.
- Text objects and Textarea objects can be validated similarly.
- You cannot return the length property of a radio group accurately from the form.elements[] array. The reason is that when you call length on a single radio object, it always returns null.
- The onSubmit() event handler of the form accepts a Boolean value indicating whether or not to process the submit request.

## Step-by-Step Instructions

1. You will be using the form found in the **ValidateMe.html** file. It would be helpful to glance through this file before you start the exercise, and take note of a few things within the file:
  - The file will look for the external .js file, **ValidateMe.js**.
  - The onsubmit event of **form1** calls the function **validateForm()**, which serves as the main function of this application.
  - By design, form objects that have names beginning with underscore are required fields.
2. Open **ValidateMe.js** and **ValidateMe.html**. In **ValidateMe.js**, create the main function, **validateForm()**. Have the function accept a parameter called **frm**, which will be used to accept a form object.

```
function validateForm(frm) {  
}
```

3. In the first few lines of the validateForm() function, define the variables **submitform = true**, **reqChar = “ ”**, and **currElement**.

**NOTE** The **submitform** variable will ultimately determine whether the form is submitted or not. The **reqChar** variable represents the first character in the name of a required field on the form. The script, when complete, will know to look for this character to make sure that that current object has a value. Finally, the **currElement** variable represents the current element in any iteration of the **forms.elements[]** array. This is important to note, because when the **ElementLoop** is stopped because of a validation error, the **currElement** will hold the information about the object that was incorrect.

```
function validateForm(frm) {
    var submitform = true;
    var reqChar = "_";
    var currElement;
```

4. Under the variable declarations, create a label called **ElementLoop** and under that, create a for loop block. Iterate the loop for the length of **frm** with the condition: **fo = 0; fo < frm.elements.length; fo++**.

```
var submitform = true;
var reqChar = "_";
var currElement;
ElementLoop:
for (fo = 0; fo < frm.elements.length; fo++) {

}
```

5. Within the loop block, create an if block. The purpose of the if block will be to catch only those elements that have the **reqChar** as the character at the first position in the string (**charAt(0)**), and to ignore any undefined elements (like the fieldset object).

```
ElementLoop:  
    for (fo = 0; fo < frm.elements.length; fo++) {  
        if (frm.elements[fo].name != undefined &&  
            frm.elements[fo].name.charAt(0) == reqChar) {  
                //Then this is an element to be validated.  
            }  
    }
```

6. The first statement in the if block must set **currentElement** to reference the currently selected form element in the iteration.

```
if (frm.elements[fo].name != undefined &&  
    frm.elements[fo].name.charAt(0) == reqChar) {  
    currElement = frm.elements[fo];
```

7. On the next line, create a switch block with four cases, and no default block. The switch condition will be (**currElement.type**). The cases represent the element types that you want to validate. For this exercise they will be as follows: **case “select-one”**, **case “radio”**, **case “text”**, and **case “textarea”**. Be sure to add a **break;** statement to each case.

```
if (frm.elements[fo].name != undefined &&  
    frm.elements[fo].name.charAt(0) == reqChar) {  
    currElement = frm.elements[fo];  
  
    switch(currElement.type) {  
        case "select-one":  
            break;  
        case "radio":  
            break;  
        case "text":  
            break;  
        case "textarea":  
            break;  
    }  
}
```

8. Right before the end of the main elements loop, add another if block with the statement **if (!submitform) { break ElementLoop; }**. This ensures that if an element does not evaluate to true, then the loop will stop iterating.

```
        case "textarea":  
            break;  
        }  
    }  
  
    if (!submitform) {  
        break ElementLoop;  
    }  
}
```

9. Underneath the main function, define the validation utility functions. There will be a validation function for each type of object caught by the switch statement in **validateForm**. Each one will accept a **frmobj**, which is also the **currElement** of the form. Name the functions **valSelect()**, **valRadio()**, and **valText()**.

```
function valSelect(frmobj) {  
}  
  
function valRadio(frmobj) {  
}  
  
function valText(frmobj) {  
}
```

10. One thing that each function has in common is a Boolean variable called **valid**. Start by defining **valid** as false; each function will eventually return the valid variable.

```
function valSelect(frmobj) {  
    var valid = false;  
  
    return valid;  
}  
  
function valRadio(frmobj) {  
    var valid = false;  
  
    return valid;  
}  
  
function valText(frmobj) {  
    var valid = false;  
  
    return valid;  
}
```

11. In the validateForm() function, each case is going to be nearly identical. Each one will define **submitform** using its corresponding utility function. For example, case “select-one” will define **submitform** by assigning it the return value of the function **varSelect(currElement)**.

**NOTE** The one exception to the rule is case “radio”. When the element in question is a radio button, **currElement** actually references an individual radio button object. The default value of an individual radio button’s length is null. In this application, you just want to make sure one of several radio buttons is selected, so you must access the length attribute of the whole radio group. This is available by establishing a direct reference to the radio group name on the form (**document.formname.radiogroupname**), which coincidentally is the same as each radio button’s name.

```
switch(currElement.type) {  
    case "select-one":  
        submitform = valSelect(currElement);  
        break;  
    case "radio":  
        submitform = valRadio(eval("document."  
            + frm.name + "." +  
            currElement.name));  
        break;  
    case "text":  
        submitform = valText(currElement);  
        break;  
    case "textarea":  
        submitform = valText(currElement);  
        break;  
}
```

12. At the very end of the validateForm() function, the code determines whether to submit or not. So beneath the Element Loop, start an if/else statement, with the condition **submitform**. If submitform is true, return true from validateForm() back to the **onsubmit** event handler that called it. If submitform is false, return false.

```
if (submitform) {  
    return true;  
} else {  
    return false;  
}
```

13. If the form validated as true, the application will generate an alert message: “**The form is complete!**”. If it is false, it will generate an alert message saying “**The selected field was not completed.**”, and then set focus to the element that is not valid. Because the ElementLoop breaks when a problem occurs, currElement will be the violating element.

```
if (submitform) {  
    alert("The form is complete!");  
    return true;  
} else {  
    alert("The selected field was not completed.");  
    currElement.focus();  
    return false;  
}
```

14. Now you can test the exercise. The **submitform** variable will be false each time, because the validation utility functions have not yet been defined. Just make sure that there are no errors, and continue on to the next exercise. The result should look something like Figure 8.

**TIP:** A form button called **shortcut** and its corresponding function have been added to the exercise files, so you can instantly fill in the form when shortcut is clicked to save some keystrokes.

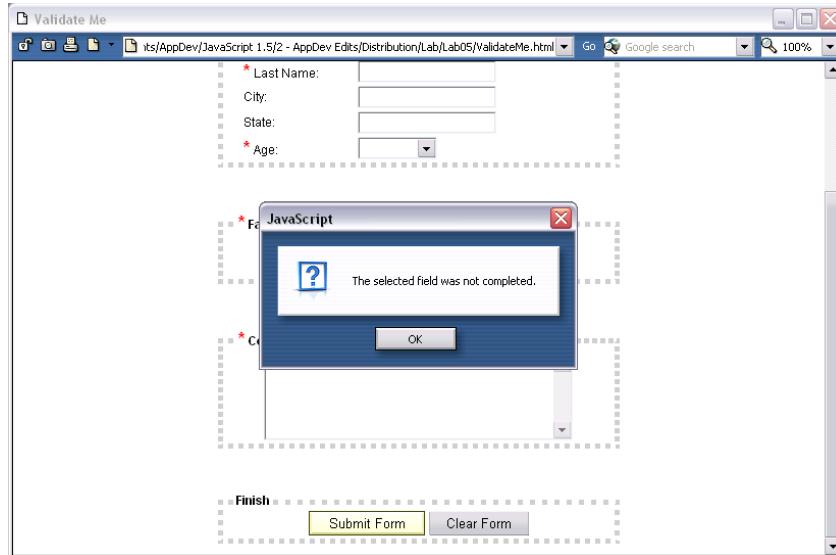


Figure 8. The result of this exercise.

# Defining Validations

## Objective

In this exercise, you will set up the actual validation for the text, textarea, radio, and select objects.

## Things to Consider

- For this example the select object's validation rule is that the current selected option must have a value, and may not contain an empty string. The default option, [Select], has an empty string for its value, and would therefore be considered invalid if it were submitted with the form.
- For the radio group, none of the radio buttons are checked by default, but one radio button must be checked by the user to be considered a valid submission.
- The default value for the text and textarea objects is an empty string, but these elements must contain some character data in order to be considered valid entries.

## Step-by-Step Instructions

1. You will start by defining the text/textarea validation function, **valText()** from the previous exercise. Create an if block just after the declaration of the valid variable. The expression for the if statement will be **(frmobj.value.length > 0)**, to test whether the length of the entry is greater than zero; if so, the valid variable must be set to true with the statement **valid = true;**

```
function valText(frmobj) {  
    var valid = false;  
    if (frmobj.value.length > 0) {  
        valid = true;  
    }  
    return valid;  
}
```

2. Next, define the select validation in the function **valSelect()**. From the chapter you should remember that a select object has a **selectedIndex** property, which will return the selected object's index position in the options array. Add an if block between the **valid** variable definition and the **return valid;** statement that has the condition **(frmobj[frmobj.selectedIndex].value.length > 0)**. This checks to make sure that the selected Index has value.

**NOTE** In Mozilla, the default value or the **[Select]** instruction-option must have a value of “” (an empty string) defined, or it will return the selected option’s label (i.e., **[Select]**) as the value.

```
function valSelect(frmobj) {  
    var valid = false;  
    if (frmobj[frmobj.selectedIndex].value.length > 0) {  
        valid = true;  
    }  
    return valid;  
}
```

3. Define the **varRadio(frmobj)** function, which will loop through the objects in the radio group parameter that is passed in. Create a for loop block with the condition **(ri = 0; ri < frmobj.length; ri++)**.

```
function valRadio(frmobj) {  
    var valid = false;  
    for (ri = 0; ri < frmobj.length; ri++) {  
        if (frmobj[ri].checked) {  
  
        }  
    }  
    return valid;  
}
```

4. Within the loop block that you created in Step 3, create an if block with the condition **(frmobj[ri].checked)**. If the current radio object in the radiogroup is checked, set the variable valid to true, using the statement **valid = true;** On the next line, add a **break;** statement to ensure that the code stops executing the loop once a checked radio object is found.

```
function valRadio(frmobj) {  
    var valid = false;  
    for (ri = 0; ri < frmobj.length; ri++) {  
        if (frmobj[ri].checked) {  
            valid = true;  
            break;  
        }  
    }  
    return valid;  
}
```

5. Now it is time to test the script. Submit the form once with only one type of form object selected (and the other types not selected) so that you can check each validation function. If a required element is not selected or filled in, it should receive focus after you click the **OK** button in the alert message dialog box, as shown in Figure 9.

**Lab 5:**  
**Form Interaction**

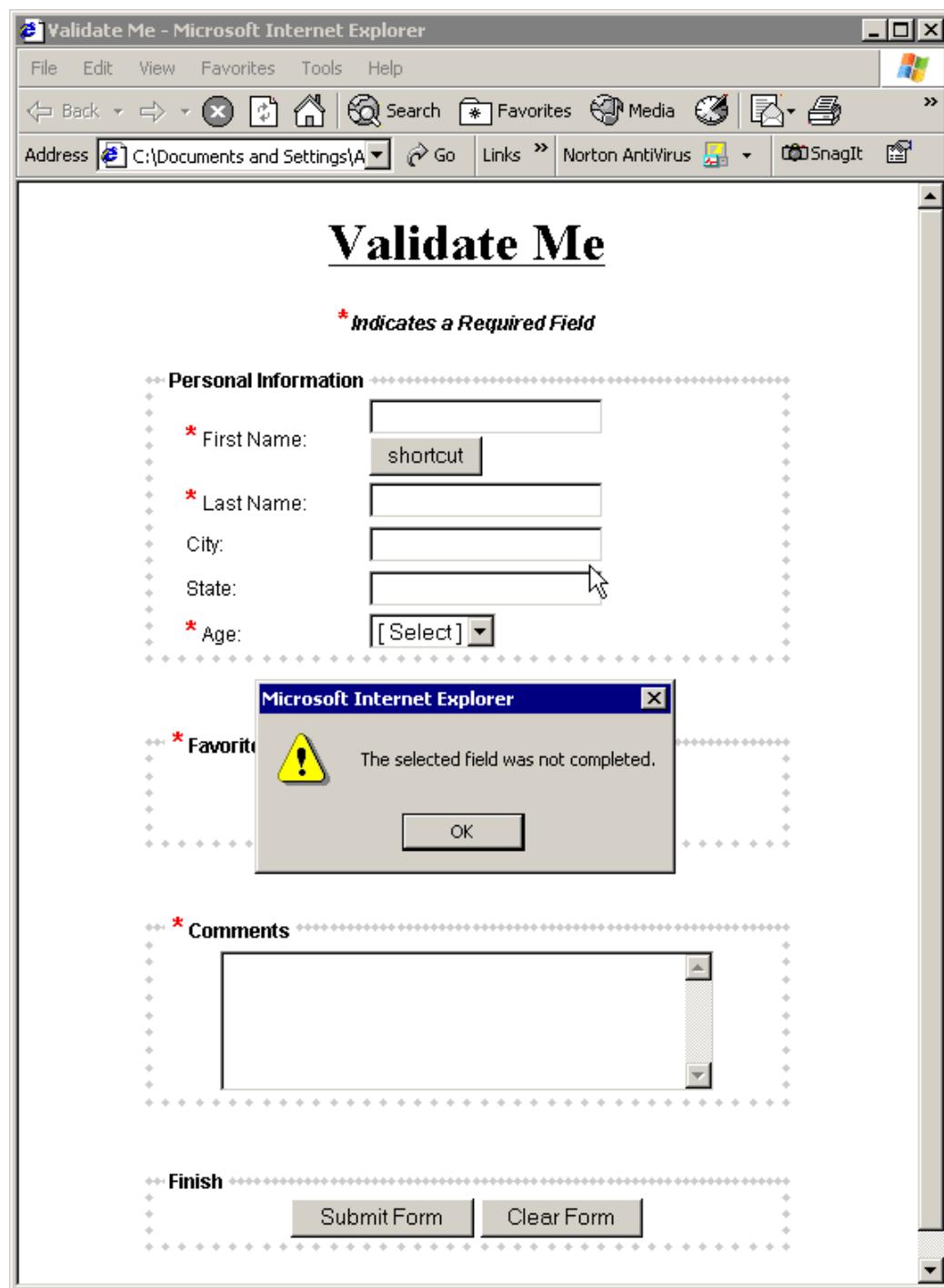


Figure 9. The Alert Message box in ValidateMe.html.

# Display Results

## Objective

In this exercise, you will parse through and display the values of the form after a successful submission by parsing through the query string for the page.

## Things to Consider

- The location object's search method returns everything after the base URL on the address bar (the question mark and everything after it).
- The form only passes a query string to the “action” page when a **GET** method is used.

## Step-by-Step Instructions

1. Define a global variable at the bottom of the **ValidateMe.js** file called **submitmsg**, and set it to reference an empty string.

```
var submitmsg = "";
```

2. On the next line, define an if block with the condition: **(window.document.location.search.length > 0)**. This tests to see if there are any parameters in the querystring, which means that the form has been submitted.

```
if (window.document.location.search.length > 0) {  
}
```

3. Next, the application will build XHTML output into the **submitmsg** variable, to be displayed on the page.

**NOTE** There is already a place in the file **ValidateMe.html** that checks **submitmsg**, and writes the contents if it has value:

```
<!-- Display Submit Results -->

<script language="JavaScript" type="text/javascript">
    //submitmsg is declared in 'ValidateMe.js'

    if (submitmsg) {
        document.write(submitmsg);
    }
</script>
```

4. On the first line of the if statement from Step 3, create a new variable called **qs** and assign it the value of **window.document.location.search**, which is the query string with the question mark at the beginning. On the next line, create an array of name-value pairs called **aQueryString**. Define it by getting the name/value pairs from the query string by first removing the question mark, then splitting the querystring at the “**&**” character.

**TIP:** So you can better see what you are working with, the query string will look like this:  
**?\_fname=111&\_lname=112&city=113&state=114&\_age=1020&\_favsite=**  
**Yahoo&\_comments=115&submit=Submit+Form**

```
var qs = window.document.location.search;
var aQueryString = (qs.substr(1,
    qs.length)).split("&");
```

The array now equals a series of name/value strings that look something like: “**\_fname=111**”. To extract the name and value, the string will have to be split again into an array using the **=** character.

5. Next you will start creating the **submitmsg** to display in **ValidateMe.html**. Start with the formatting, which you can copy from the following code snippet:

```
submitmsg = "<fieldset class=\"fieldset\"><legend  
class=\"text\"><b>";  
submitmsg += "Submission Results</b></legend><div  
align=\"center\">";  
submitmsg += "<font class=\"text\" color=\"green\">";  
  
// INSERT NAME / VALUES HERE  
  
submitmsg += "</font></div></fieldset>";
```

6. Add a for loop where the comment **// INSERT NAME / VALUES HERE** is located, with the condition (**al = 0; al < aQueryString.length; al++**). This loop will iterate through the name-value pairs that were extracted into the **aQueryString** array. For the first statement in the loop, split up the name and value of the current pair into another temporary array, using the statement **a = aQueryString[al].split("=");**.

```
for (al = 0; al < aQueryString.length; al++) {  
    var a = aQueryString[al].split("=");  
  
}
```

7. For the second statement in the loop, output the name and value of the current index in the array in XHTML. Remember, **a[0]** references the name and **a[1]** references the value:

```
for (al = 0; al < aQueryString.length; al++) {  
    var a = aQueryString[al].split("=");  
    submitmsg += ("<b>" + a[0] + "</b> is equal to  
    <b>" + a[1] + "</b><br/>");  
}
```

8. Test the completed project by launching **ValidateMe.html**. Once you have completed the form and submitted it, all the values should be shown on the page, with the complete form resembling the one in Figure 10.

**Lab 5:**  
**Form Interaction**

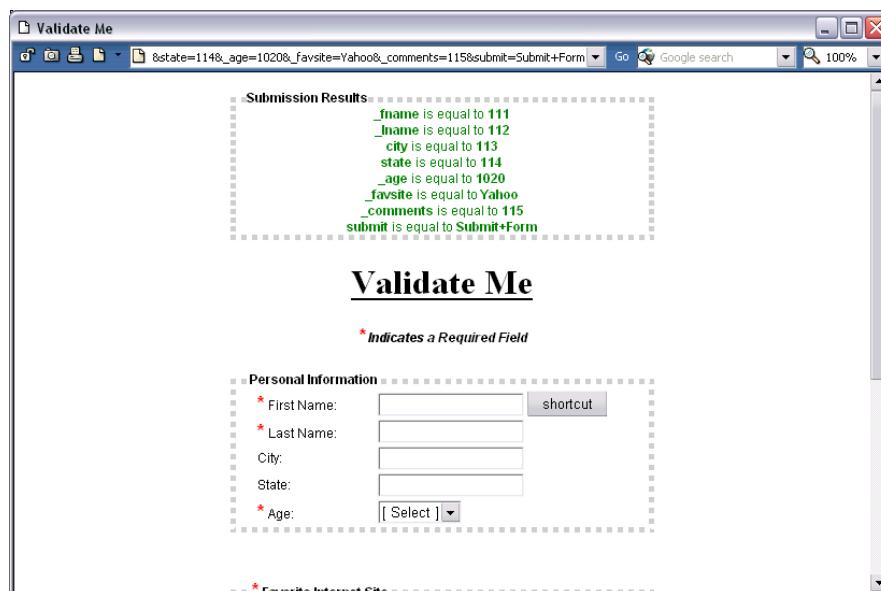


Figure 10. The final page after it has been submitted successfully.

# Built-In Objects

## Objectives

- Understand how the most common JavaScript objects are used.
- Discern the available information in the Navigator object.
- Discover how to use the Date object.
- Calculate with the Math, Number, and Boolean objects.
- Learn to traverse String objects.
- Express yourself with RegExp.

# String Object

At this point, you should be familiar with basic string manipulation, such as concatenation, substrings, and case modification. There are, however, more advanced actions that String objects can perform, which you will see in the following sections.

Keep in mind that, in JavaScript, a string variable, a String object, and a string of characters in double quotes are all converted to String objects behind the scenes. As a result, you can use any of the String object method calls from each of these string variations.

## Prototype Properties and Methods

String prototypes allow you to create your own custom string methods, which can be called just like any of the String object's built-in methods. Using string prototypes for custom functionality is very convenient, and they are generally easier to follow in code than when strings are passed to functions.

In string prototyping, you first create a function that manipulates and returns a string, and then you create an alias to that function with a string prototype method. After the prototype is defined in your script, any string can call your custom prototype method while the script is in scope.

In the following example, a function called formatHeading() is created to add <h3> heading tags around a string. Then a string prototype method called headerMe() is defined that references the formatHeading() function. Every string created after the prototype definition can now use the headerMe() method to surround itself with <h3> tags. The method is linked to the defining function, which uses the keyword *this* to represent the string that is calling it.

```
function formatHeading() {  
    return "<h3>" + this.toString() + "</h3>";  
}  
  
String.prototype.headerMe = formatHeading;  
  
document.write("Hello World!".headerMe());
```

In this example, String.prototype.headerMe is defined as the function ‘formatHeading()’. When called by the “Hello World!” string, the function executes. The function recognizes that the keyword ‘this’ refers to the string “Hello World!” because “Hello World!” is the String object that called it.

Of course, this also works for a string variable:

```
var hello = "Hello World Variable!";
document.write(hello.headerMe());
```

Or a String object:

```
var helloObj = new String("Hello World Object!");
document.write(helloObj.headerMe());
```

## charAt()

Use the charAt() method for accessing a character that appears at a specific position within a string. The charAt() method takes an integer as a parameter; this integer indicates the index position of the character that you want to retrieve from the string.

```
var myString = "Hi there!";
var myChar = myString.charAt(0); // myChar equals 'H'
```

You probably noticed in this example that the index value passed to charAt() was 0. JavaScript can handle a string as a zero-based array of characters (which means that the first position in the array index is zero).

One of the ways you can use charAt() is to mark special fields in a form with an exclamation point as the first character, which you can then find and evaluate using the charAt() method:

```
<script language="JavaScript">
<!--
function clicked(btn) {
    if (btn.name.charAt(0) == "!")
        alert("You are the real button.");
    } else {
        alert("You are an imposter!!");
    }
}
//-->
</script>

<input type="button" name="!original" value="Click me!" onclick="clicked(this)"/>
<input type="button" name="impostor"
       value="No, No, Click me!" onclick="clicked(this)"/>
```

Another use for the charAt() method is to reverse the order of characters in a string. Using the charAt() method, you can loop backwards through the original string and build a new string using the characters from the first string. Since strings are zero-based character arrays, you must use (string.length - 1) to determine the position of the last character in the string, for use as the index value in the charAt() method.

```
var dlroWh = "!dlrow ,ereht olleH";
var hWorld = "";
for (si = (dlroWh.length - 1); si >= 0; si--) {
    hWorld += dlroWh.charAt(si);
}
alert(dlroWh + " became " + hWorld);
```

## charCodeAt()

In some situations, you may need to generate the ASCII codes for string characters. If you send a character to a cgi application that requires ASCII codes instead of characters, you can use the String object's charCodeAt() method. This method works just like charAt(), but returns the ASCII code for

the given character at the specified index position. For example, the `charCodeAt()` method call in the following code example returns the ASCII value 98.

```
alert("library".charCodeAt(2)); // result for 'b' is 98
```

## **fromCharCode ()**

Alternately, if you wish to convert a row of ASCII characters to a string, you would use the `String.fromCharCode([asciiChar#_1], [asciiChar#_2], etc)`:

```
alert(String.fromCharCode(72, 101, 108, 108, 111));
//result is 'Hello'
```

## **String.match(RegExp)**

See

*StringRegExp.html*

The static method, `String.match()` can be used to determine how many times a particular piece of a string might appear. When the `match()` method finds a match, it dumps the results into an array and continues adding to that array as it finds matches. If the `match()` method finds no matches, it returns null. You will learn more about regular expressions, or the `RegExp` object later in this chapter, but the following example shows how a simple version works:

```
var statement = "Indeed I need to feed with seed.";
var arrayofEeds = statement.match(/eed/g);
var count = 0;
for (i = 0; i < arrayofEeds.length; i++) {
    count++;
}
document.write(count);
//Result of count: 4
```

When the regular expression `/eed/g` is passed to the `match()` method, it finds matches in the substrings *indeed*, *need*, *feed*, and *seed*. Notice that the regular expression is not in quotes, and is also trailed with a `/g`. The `/g` character specifies that you want it to perform a global match, which tells the method to search the entire string. If you omitted the `/g`, the result of `count` would be 1, because it would stop after the first instance of “eed” in the word *indeed*.

## String.replace(RegExp, string)

See

[StringRegExp.html](#)

The replace method is similar to match, except that it finds the values and replaces them with whatever you specify:

```
statement = "You c, I have a c+ in that class.";
var edited = statement.replace(/c/g, "see");
document.write(edited);
//Result: "You see, I have a see+ in that seelass."
```

Unfortunately, the expression found every *c* in the statement! You can fix this with another type of RegExp character, \b. If you surround your expression with \b, or boundaries, the expression looks only for a “c” that is not attached to a word:

```
var revised = statement.replace(/\bc\b/g, "see");
document.write(revised);
//Result: "You see, I have a see+ in that class."
```

This is a little better. Punctuation and special characters are not considered words, so the boundary modifier ignores the + in *c+*. You will learn more about using regular expressions later in the chapter.

## String.split("delimiter", [limit int]) or (RegExp)

See [Prototype Example.html](#)

The last element to cover for strings is the split() method. The split() method splits a string into an array of substrings, creating a new substring wherever the given delimiter character exists in the string. If a limit integer, such as 5, is specified, then the split() method will create an array that contains substrings for the first five delimiters it encounters in the string.

The split() method is quite helpful if you need to parse name-value pairs from the querystring of the request URL:

```
var query =  
    "txt1=Fred&txt2=Bob&txt3=Frankford&txt4=David";  
var aNames = query.split("&", 3);  
for (i = 0; i < aNames.length; i++) {  
    document.write(aNames[i] + "<br/>");  
}
```

In the preceding example, you would get an array containing the first three name-value pairs, txt1=Fred, txt2=Bob, and txt3=Frankford.

While this is a good start, you may simply need the values but not the names of the form elements that were included in this query string. This is where things get a bit more complicated. You could cull out the values by splitting the name-value pairs again using = as a delimiter and access every other element in the array to get just the values. Another method would be to use a more complex regular expression to pull out only the values you need, as shown in the following example:

```
var rege = /txt\d=|&txt\d=/  
aNames = query.split(rege);  
for (b = 0; b < aNames.length; b++) {  
    document.write(aNames[b] + "<br/>");  
}
```

The regular expression from right to left looks for the letters txt\d=, where the \d characters represent a single digit. The pipe character is the OR operator. It tells the method to also look for characters matching the expression &txt\d=. In English, you would read this expression as: “Provide any value that appears after the characters txt or &txt when they are followed by a single numeric character and an equal sign”.

Regular expressions and the regular expression object are covered in more detail later in this chapter.

# Date() Object

The Date object was not widely accepted in the earlier days of JavaScript; it was buggy and had many cross-platform issues. Fortunately, the Date object in JavaScript 1.5 is in good form and can provide a powerful asset to applications that need to implement Date/Time functionality. Before working with the Date object, you need a basic understanding of the time standard, Greenwich Mean Time (GMT), which is also known as Coordinated Universal Time or (UTC).

A computer's internal clock normally is set to GMT time. Somewhere in your operating system's setup, you have the ability to set your local time zone. Regardless of your local time zone, the computer's clock stays the same. Once the operating system knows the offset for your particular time zone, it displays the time based on GMT plus that offset.

In JavaScript, date and time is set similarly. The Date object is initialized using the time information from the client browser's operating system, rather than from the Web server. When the Date object is required to output time information to the browser, it uses the localized time, which includes the offset from GMT. If the client computer's clock is inaccurate then the time displayed by the script will reflect this inaccuracy in the browser.

Despite their name, Date objects are created with information about both the date and the time. At the time a Date instance is created, an exact snapshot of the main, static Date object is created right down to the millisecond. So while you are working with that Date instance, keep in mind that it is not ticking or counting up the seconds into the future. Rather, it is an object that contains the specific information from the time it was created from the main Date object.

Creating a Date object is a simple matter of using the new keyword and calling the Date object's default constructor:

```
var rightnow = new Date();
```

When the Date() constructor executes, the Date object's snapshot in time is created. The *rightnow* reference provides valuable information about the current system date and time.

<b>WARNING!</b>	JavaScript months are treated like a zero-based array, and start at 0. When using Date objects, you must remember that the month values run from 0 – 11, where January is 0 and December is 11.
-----------------	---

See  
[DateObject.html](#)

In the following example, the *rightnow* date object is used to display the full date to a client browser:

```
var month = rightnow.getMonth();
var day = rightnow.getDate();
var year = rightnow.getYear();
var cdate = "";
cdate += "Current Date: (m/d/yyyy)<br/>";
cdate += month + "/" + day + "/" + year;

document.write(cdate);
```

**NOTE** If you run this script in Opera 7 you will notice that the year returned is not quite right. For 2003, Opera will display 103. This happens because the baseline year in Opera's implementation of JavaScript starts at 0 instead of 1900. To fix this, your script should determine whether the year value has less than four digits, and if so just add 1900 to it to get the correct year.

Furthermore, you can display current time information using the time methods of the date object:

```
var ctime = "";
var hours = rightnow.getHours();
var minutes = rightnow.getMinutes();
var seconds = rightnow.getSeconds();
var milliseconds = rightnow.getMilliseconds();
ctime += "Current Time: (hh:mm:ss:ms)<br/>";
ctime += hours + ":" + minutes + ":" + seconds +
        ":" + milliseconds;
document.write(ctime);
```

Once the code is running, you can refresh the browser several times to see the time update gradually.

In addition, if you are concerned with working in GMT time, you can extract that information using the Date methods *getUTCMonth()*, *getUTCDate()*, *getUTCDay()*, *getUTCHours()*, *getUTCMinutes*, *getUTCSeconds()*, and *getUTCMilliseconds()*.

There is an easier way to display time and date strings, as JavaScript is equipped with handy functions to do that for you. The first type of function displays the date and time in a string format determined by the local operating system:

```
document.write(rightnow.toLocaleDateString() + "<br/>");  
document.write(rightnow.toLocaleTimeString() + "<br/>");  
document.write(rightnow.toLocaleString() + "<br/><br/>");
```

The second type of function allows the browser to choose the format in which to display them:

```
document.write(rightnow.toDateString() + "<br/>");  
document.write(rightnow.toTimeString() + "<br/>");  
document.write(rightnow.toString() + "<br/><br/>");
```

And the final option displays them in GMT time:

```
document.write(rightnow.toUTCString() + "<br/>");
```

In the Mozilla and Internet Explorer browsers, the full month and day name displays in the local versions. You cannot rely on this formatting, however, because other browsers may not display them in the same way. Figure 1 illustrates the output of the Date functions on the Mozilla browser.

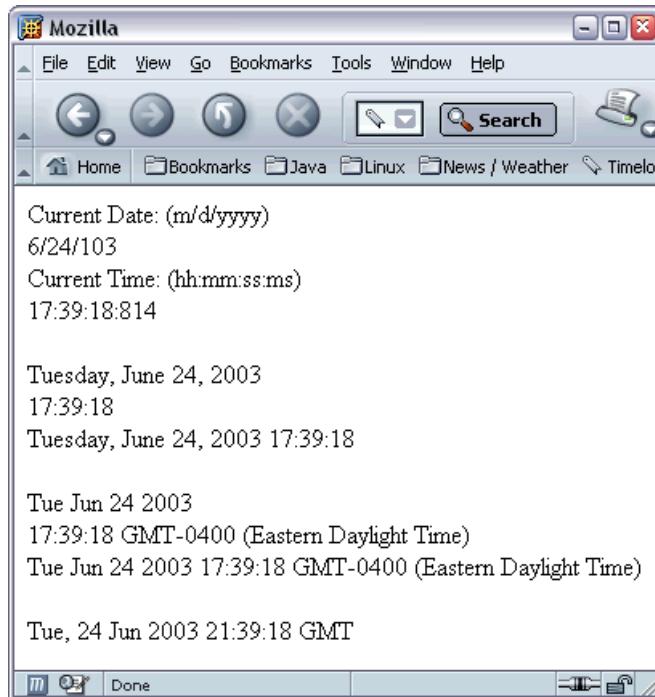


Figure 1. The date display in the Mozilla browser.

In Figure 2 you will notice that the years render differently then in Mozilla (or most other browsers) when the script is run in Opera 7.

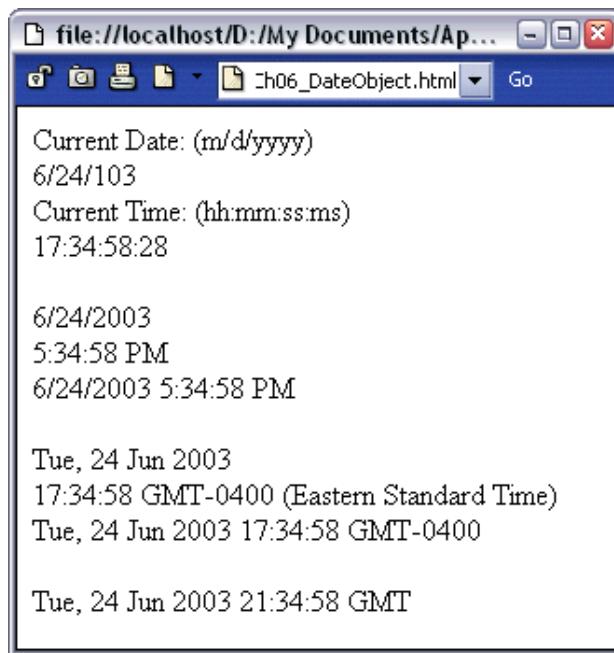


Figure 2. The date display in the Opera browser.

To get a formal date string that provides the same output across all browsers, you must create two arrays to hold the formal values for the names of the days and months:

```
var days = new Array("Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday", "Friday");

var months = new Array("January", "February",
    "March", "April", "May", "June",
    "July", "August", "September",
    "October", "November", "December");
```

Since the Date object's `getDay()` and `getMonth()` methods are zero-based, the integer values that they return will match the corresponding positions in the day and month arrays. You can use those integers as array index values to get the formal day and month names in your script.

## Working with Other Dates

When you work with applications, such as a calendar, you may need to work with Date objects that are not necessarily the current snapshot in time. Fortunately, JavaScript offers some alternative constructors and methods to accommodate the creation of “non-current” Date objects.

With these alternate constructors you can create a Date object based on the date and time of your choosing. The syntax for the constructor is as follows:

```
new Date(<parameters>); // parameters determine date
```

Table 1 shows the various constructor parameters you can use to create the new Date object:

Possible Date Constructor Parameters
milliseconds
“Month dd, yyyy”
“Month dd, yyyy hh:mm:ss”
yy, mm, dd, hh, mm, ss
yy, mm, dd

Table 1. Date constructor parameters.

To create a Date object for June 24, 2003, you could use the following code:

```
dateobj = new Date("Jun 24, 2003")
```

Another way to customize a Date object is to modify its values using the object's set methods. The Date object has quite a few built-in methods for setting its attributes:

```
setYear(year);
setFullYear(year);
setMonth(0 - 11);
 setDate(1 - 31);
setHours(0 - 23);
setMinutes(0 - 59);
setSeconds(0 - 59);
setMilliseconds(0 - ...);
setTime(0 - ...);
setUTCFullYear(1970 - );
setUTCMonth(0 - 11);
setUTCDate(1 - 31);
setUTCDay(0 - 6);
setUTCHours(0 - 23);
setUTCMinutes(0 - 59);
setUTCSeconds(0 - 59);
setUTCMilliseconds(0 - ...);
```

# **setInterval() and setTimeout()**

If you have used the Web to any extent, you have almost certainly come across some kind of JavaScript clock, animation, or other timed trick. The setInterval() or setTimeout() methods are most likely the engine behind the timed effect.

For example, if you create some sort of function or algorithm that must run over and over again, or you script a live clock that does not require a refresh from the server, then you would want to use the setInterval() method.

Alternately, if you have an application that has timed events that fire at a specific interval after the page loads (for instance, an annoying pop-up that appears exactly two seconds after the page loads) then setTimeout() is the way to go.

## **setInterval**

The setInterval() method is useful for creating live action events, animation, or live clocks. You define an interval by assigning it to a variable, which is then passed as a parameter to the setInterval() method. Once setInterval() is called, the variable name is used as a unique identifier to the timer. If you want to stop the events of the timer, simply call JavaScript's clearInterval() method and pass it the variable name of the original interval that you passed to the setInterval() method.

*See [Interval\\_Timeout.html](#)*

The following example displays a simple clock. If your system has less than a Pentium II processor, you might want to set the setInterval() value to 1,000 instead of 1, as the update might slow your computer down too much otherwise.

```
<script language="JavaScript">
<!--
var frmobj;
var interval = null;
function startClock(obj) {
    frmobj = obj;
    if (!interval) {
        interval = setInterval(updateClock, 1);
        //Eliminates one second start delay.
        UpdateClock();
    } else {
        alert("Clock is running.");
    }
}

function stopClock() {
    clearInterval(interval);
    frmobj.value = "Show Clock";
    interval = null;
}

//Private Function for the Clock Updates
function updateClock() {
    var now = new Date();
    frmobj.value = now.toLocaleTimeString() +
        " -- milli:" + now.getMilliseconds();
}
//-->
</script>

<form name="form1">
    <input type="button" name="clock" value="Show Clock"
        onClick="startClock(this)"/>
    <input type="button" name="click" value="Stop Clock"
        onClick="stopClock()"/>
    <br/><br/>
    <input type="button" name="timer"
```

```
        value="Show 1 Minute Timer"
        onClick="startClock(this)"/>
<input type="button" name="click2"
        value="Reset Timer" onClick="stopClock()"/>
<input type="button" name="click3"
        value="Stop Timer" onClick="stopClock()"/>
</form>
```

Figure 3 displays the results of executing the previous script.

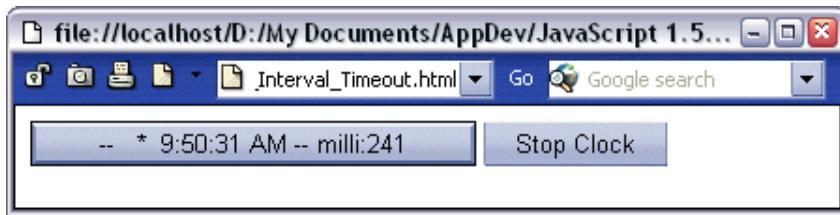


Figure 3. The clock running in Opera 7.

## **setTimeout**

The `set Timeout()` method is handy for applications that need to execute methods at a specific time. For example, if you need the page to refresh every minute, you could set the timeout at one minute or 6,000 milliseconds, and then execute a function to refresh the page. A timeout could also be used to create the clock in the `setInterval()` example. In that scenario, you would create a timeout to call an update time function every second, which in turn, would create another timeout for the next minute, and so on.

As with `setInterval()`, in `setTimeout()` you declare a variable to hold the timeout value and pass it as a parameter to the `setTimeout()` method. That variable name can be used as a unique identifier for the timeout mechanism. In addition, you can clear the timeout by calling the JavaScript method `clearTimeout()` and passing it the timeout variable.

*See [Interval\\_Timeout.html](#)*

The following example constructs a simple countdown timer. The `startTimer()` function gets the number of seconds from the text object that is sent in, starts the timeout mechanism, and calls `updateTimer()` to maintain the rest of the countdown process. The `updateTimer()` function decrements the `cnt` variable until it reaches zero.

```
<script language="JavaScript">
<!--
var timerID = null;
var timerobj;
var cnt = 0;
var save = 0;

function startTimer(obj) {
    timerobj = obj;
    if (!timerID) {
        cnt = parseInt(timerobj.value);
        save = cnt;
        document.bgColor = "green";
        timerID = setTimeout("updateTimer()", 1000);
    }
}

function stopTimer() {
    clearTimeout(timerID);
    timerID = null;
    timerobj.value = save;
    document.bgColor="white";
}

function updateTimer() {
    if (cnt == 0) {
        stopTimer();
        alert("Finish!!!");
    } else {
```

```
        cnt --;
        timerobj.value = cnt;
        timerID = setTimeout("updateTimer()", 1000);
    } } //-->
</script>
<form name="form1">
<input type="text" name="timeval" value="10"
       size="2"/>&nbsp;secs&nbsp;;
<input type="button" name="btntimerstart"
       value="Start Timer"
       onClick="startTimer(document.form1.timeval);"/>
<input type="button" name="btntimerstop"
       value="Stop Timer" onClick="stopTimer();"/>
</form>
```

Figure 4 displays the results of running the countdown timer example.

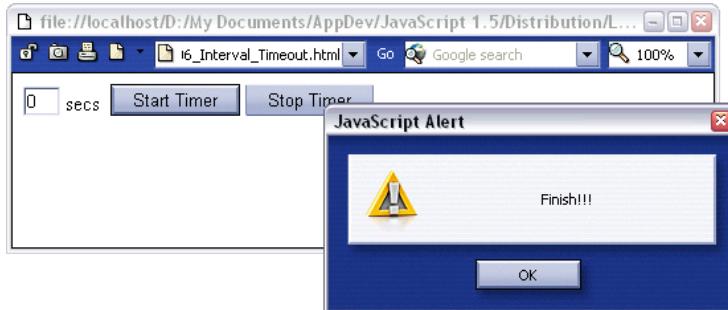


Figure 4. The timer running in Opera 7.

Now you should be able to figure out for yourself how to use these functions to create unique animations or to enhance the usability of your pages.

# Math Objects

In most programming languages, mathematical operations are central to complex algorithms and procedures. JavaScript provides a variety of Math objects, which give a scripter access to a wide range of mathematical functions.

## The Number Object

A Number object in JavaScript can refer to one of two things: an integer or a floating-point value. If you are familiar with the broader range of numeric data types available in a programming language such as C or Java, this may appear to be a limitation, but from a scripting perspective these data types are enough to get the job done.

In most of your scripts you will deal primarily with JavaScript number variables. However, in some cases the JavaScript Number object can come in handy. Perhaps the most useful aspect of the Number object is its prototype method, which works much like the String object's prototype method. It enables you to create your own custom methods, which can be called from any Number object. The following example defines a function, dividebytwo(), and then creates a new Number prototype called getHalf() that aliases it:

```
function dividebytwo() {  
    return (this / 2); // 'this' references Number object  
}  
  
Number.prototype.getHalf = dividebytwo;  
  
ten = new Number(10);  
alert(ten.getHalf()); // result is 5
```

While you are able to call prototype methods directly from any variation of a String object, this is not the case with numbers. You can only call prototype methods on a Number object that you have created with the new Number() constructor, or a variable that points to a number, but not an actual number.

```
function getTimes() {
    return this * this;
}

Number.prototype.timesMe = getTimes;

var ten = 10;
ten.timesMe // result is 100

var twenty = new Number(20);
twenty.timesMe // result is 400

30.timesMe // generates an error
```

A browser imposes some restrictions on the Number object, which are evident in the static properties of the Number object itself. These properties are MAX\_VALUE, MIN\_VALUE, NEGATIVE\_INFINITY, and POSITIVE\_INFINITY. The max and min value properties return the maximum and minimum number that this browser's JavaScript can handle. If a number falls above or below these max/min boundaries, JavaScript sets the values to Number.POSITIVE\_INFINITY and Number.NEGATIVE\_INFINITY, respectively. This is helpful to know when you test whether a browser can handle very complicated mathematical operations.

Figure 5 shows the static Number object limitations in the Opera 7 browser.

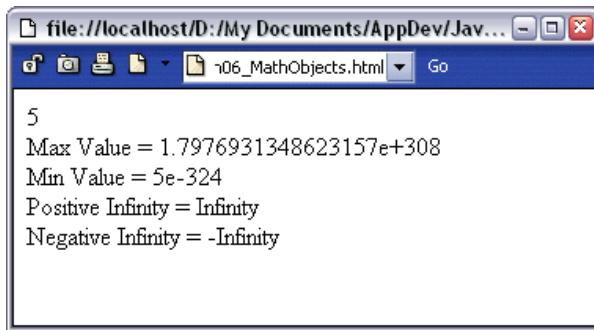


Figure 5. Number static properties results in Opera 7.

Table 2 lists the methods that can be called from a Number Object.

Methods of the Number Object
toExponential()
toFixed()
toLocaleString()
toPrecision()
toString()
valueOf

Table 2. Number object methods.

## The Boolean Object

The Boolean object is rarely used, due to the simplicity of the Boolean values used in the average scripts. One case in which you might want to create a new Boolean object is when you are evaluating an expression that will then be assigned to a variable, as in the following example:

```
var i = 1;
var result = new Boolean(i == 0);
// result = false
```

## The Math Object

The Math object is the heart of any mathematical functions in JavaScript that perform more than just simple arithmetic. This object contains many methods to make common mathematical tasks much easier for the scripter. The Math object also includes an array of properties that define mathematical constants, such as PI.

The Math object has convenient static properties and methods that can be used in statements, much like any other JavaScript object:

```
8 * Math.PI; // a static value for PI
Math.round(8.33); // an automatic rounding function
```

## Working with Random Numbers

The `Math.random()` method does exactly what it says: it returns a random number. If you call `random()` without passing it any parameters, it returns a floating-point value between 0 and 1. To get a random integer value, you can use the `random()` method in conjunction with the `Math` object's `floor()` method, which takes any number as a parameter and returns the next integer that is less than or equal to that number.

To generate a random number between 0 and another number, multiply the result of `Math.random()` by a number representing the highest possible integer value that can be generated, then pass the result to the `Math.floor()` method:

```
var num = 11862.17;  
Math.floor(Math.random() * num); // result: 0 to 11862
```

A simple way to get a random number within a range starting with one (for example, an hour in the day) works as follows:

```
Math.floor(Math.random() * 12) + 1
```

If the lower number is any higher, this formula can break, though. You must test your calculations to make sure the users of your script will get a “legal” random number from a custom range.

## Math Object Methods and Properties

Table 3 lists the available methods of the `Math` Object, as a reference.

Math Object Method	Description
Math.abs(num);	num's Absolute Val
Math.acos(num);	num's arc cosine
Math.asin(num);	num's arc sine
Math.atan(num);	num's arc tangent
Math.atan2(num, num);	The angle of polar coordinates for the two numbers
Math.ceil(num);	The next integer that is greater than or equal to num
Math.cos(num);	num's cosine
Math.exp(num);	Euler's constant to the power of num
Math.floor(num);	The next integer that is less than or equal to num
Math.log(num);	Natural logarithm, base e, of num
Math.max(num, num);	The greater of two nums
Math.min(num, num);	The lesser of two nums
Math.pow(num, num2);	num1 to the power of num2
Math.random();	Random floating-point number between 0 and 1
Math.round(num);	num + 1 when the value is greater than or equal to 0.5, otherwise num.
Math.sin(num);	Sine of num
Math.sqrt(num);	Square root of num
Math.tan(num);	Tangent of num

Table 3. Methods of the Math object.

Table 4 lists the available properties of the Math object, as a reference.

Math Object Property	Value	Description
E	2.718281828459045091	Euler's constant
LN2	0.6931471805599452862	Log of 2
LN10	2.302585092994045901	Log of 10
LOG2E	1.442695040888963387	Log base-2 of E
LOG10E	0.4342944819032518167	Log base 10 of E
PI	3.141592653589793116	pi
SQRT1_2	0.7071067811865475727	Square root of 0.5
SQRT2	1.414213562373095145	Square root of 2

Table 4. Properties of the Math object.

# Regular Expressions and the RegExp Object

The use of regular expressions in JavaScript offers tremendous benefits when you work with validation and search type functions. Regular expressions can save you a substantial amount of time compared to writing complex character evaluation logic to achieve the same functionality. The tradeoff is that regular expressions take time to master, and can be complicated and confusing for beginning scripters if they have never used them. However, the amount of time and typing you can save by using regular expressions will become quite apparent the more you use them.

In JavaScript, a regular expression starts and ends with a forward slash; the regular expression itself appears between the slashes. As you saw earlier in the chapter, the syntax for creating the expression is almost a language unto itself. It consists of a series of key words and slashes, brackets, parentheses, and other characters that combine to form an expression that searches for specific patterns.

The most basic use for a regular expression is to find certain patterns within a given string, as illustrated earlier in the chapter. More complicated applications might include using regular expressions to validate the format of a date that a user entered in a text field.

Harnessing this power in JavaScript can be a great time saver, so do your best to master these skills. Though this chapter will touch briefly on the RegExp object and the use of regular expressions in general, there are also plenty of references available for using regular expressions in JavaScript.

## Simple Pattern Expressions

If you thought that you could perform pattern matching with the String object's `indexOf()` and `lastIndexOf()`, you would be correct. This approach would be sufficient for many of the simple pattern matching examples shown in this section. However, as the patterns you deal with become more complex, those methods require much more work and become quite tedious to write and cumbersome to maintain.

As stated earlier, a regular expression is contained within two forward slashes. The simplest expression is designed to find a particular substring within another string. You can also specify whether you want the expression to parse the entire string for every instance of the substring, or simply find the first occurrence of the substring. Notice that the expression pulls out every instance of the string you define:

```
var d = document;  
var message = "I can see the farmer seeding seeds.";  
var exp = /see/; //Notice the expression uses no quotes!  
var arrayMessage = message.match(exp);  
  
d.write(arrayMessage.length);  
//result = 1;
```

The result from the preceding script is 1—but not because the regular expression determined that the characters *see* appeared as a separate word only once. Rather, it occurred because in a simple expression like /*see*/, the expression quits after it finds the first match.

Regular expressions can utilize many keywords. The first you will learn is *g*, which stands for global. This keyword ensures that the expression checks for any and all occurrences of the substring that exists in the entire string:

```
var d = document;  
var message = "I can see the farmer seeding seeds.";  
var exp = /see/g;  
var arrayMessage = message.match(exp);  
  
d.write(arrayMessage.length);  
//result = 3;
```

Note that the previous expression found all three occurrences of the characters *see*, even when they were part of other words.

Now suppose that you only want to capture the characters *see* when they appear by themselves, and exclude those occurrences that appear in the words *seeds* and *seeding*. This calls for another special keyword, \b, which stands for boundary. Surrounding the search characters with the \b keyword instructs the expression to match a substring only if it has empty spaces on either side, or has only punctuation marks or special characters like + on either side. Consider the following example carefully:

```
message = "'I can see the farmer seeding seeds.'," +
    "she said. 'I see,' replied pig.";
exp1 = /see/g;
arrayMessage = message.match(exp1);
d.write(arrayMessage.length + "<br/>");  

//result = 4

exp2 = /\bsee\b/g;
arrayMessage = message.match(exp2);
d.write(arrayMessage.length + "<br/>");  

//result = 2
```

Note that by using exp2, which uses the boundary keyword on either side of the search characters, the expression finds only the two matches where the characters *see* stand by themselves.

In addition to the \b keyword, many other keywords or special characters exist that you can use to fine-tune your regular expressions, as shown in Table 5.

Keywords for Regular Expression Matching		
Keyword	Represents	Description
\B	Non-word boundary	Matches a word with word boundaries: /\Blib/ will match <i>glib</i> but not <i>liberty</i>
\b	Word boundary	Matches a word without boundaries: /\blib/ will match <i>liberty</i> but not <i>glib</i>
\D	Non-numeric	/part\D/ will match <i>partA</i> but not <i>part1</i>
\d	Numeric	/part\d/ will match <i>part1</i> but not <i>partA</i>
\S	Single white non-space	/Macho\Sman/ will match <i>Macho-man</i> but not <i>Machoman</i> or <i>Macho man</i>
\s	Single white space	/Macho\sman/ will match <i>Macho man</i> but not <i>Machoman</i> or <i>Macho-man</i>
\W	Not a letter, number, or underscore	/Dave\W/ matches <i>Dave++</i> but not <i>Dave123</i> , <i>DaveA+</i> , or <i>Dave_</i>
\w	Letter, number, or underscore	/Dave\w/ matches <i>Dave123</i> , <i>DaveA+</i> , and <i>Dave_</i> but not <i>Dave++</i>
.	Any single character, with the exception of a new line	/../ will match any two characters as long as they are in one line
[^...]	Set of non-characters	/[^sb]ee/ will match <i>tee</i> but not <i>see</i> or <i>bee</i>
[...]	Set of characters	/[sb]ee/ will match <i>see</i> or <i>bee</i> but not <i>tee</i>

Table 5. Regular expression keywords.

**NOTE** You can use brackets to define complex variations in a regular expression, to fine-tune your match criteria. For example, you can specify [0-9] or [a-zA-Z], which means that the value can be zero through 9 or, in the second example, a through z and A through Z.

You can see that it can be difficult to read regular expressions until you become familiar with the expression keywords and how they work. If your work requires you to match complex character strings, however, it is well worth the time you must invest to learn them.

It is important to note that you can combine the modifiers, like g and i, at the end of an expression. The modifier i specifies a case insensitive pattern. And similarly, if you add gi, you specify a global case insensitive pattern.

```
message = "'I can see the farmer seeding seeds.'," +
    "she said. 'I See,' replied pig.";
exp = /\bsee\b/g;
arrayMessage = message.match(exp);
d.write(arrayMessage.length + "<br/>");
//result = 1

//Using 'i' modifier
exp = /\bsee\b/gi;
arrayMessage = message.match(exp);
d.write(arrayMessage.length + "<br/>");
//result = 2
```

**TIP:** Since each regular expression keyword consists of a single character, it helps if you focus on each individual character in an expression as you read it, to determine the function that each one will perform.

You can also use regular JavaScript operators in regular expressions to string a series of search characters together. Note that the OR operator for regular expressions is simply a single pipe character, as opposed to JavaScript's double pipe character (||).

## The RegExp Object

**WARNING!** Due to limited support for the RegExp object in the Opera and Internet Explorer browsers, this section is only targeted for the Mozilla 1+ browsers. Opera does not support the RegExp object at all, whereas Internet Explorer supports a few of the less useful properties, such as leftContext and rightContext. For this reason, the scripts in this section will only function properly in Mozilla and Mozilla variants, such as Netscape and MozillaFirebird.

To fully tap the power of regular expressions, you must understand how the static RegExp object works. When you create or define a regular expression, you are basically firing a new RegExp() command, which returns an instance of a regular expression object. Each regular expression object has its own

properties and methods that you can call. At the same time, a static RegExp() object also exists, complete with its own properties and methods that work with the current window.

A number of properties are available when you create a regular expression, as shown in the following example:

```
exp = /\bsee\b/gi;

d.write(exp.source + "<br/>");  
// displays the source expression, or "\bsee\b"

d.write(exp.global + "<br/>");  
// displays the expression's "global-ness"; true for 'g'

d.write(exp.ignoreCase + "<br/>");  
// displays whether expression ignores case; true for 'i'

d.write(exp.lastIndex + "<br/>");  
// displays initial index value of 0
```

The RegExp static object also has a few properties, as shown in Table 6.

RegExp Property	Description
input	Displays the last string evaluated.
multiLine	Returns true or false depending on whether the input is more than one line.
lastMatch	Returns the index position of the last match that was found in the input.
lastParen	Displays the last value matched that was in parentheses in the expression.
leftContext	Returns the value of input that was to the left of the last match.
rightContext	Returns the value of input that was to the right of the last match.
\$1 - \$9	(These are for storage from back references, which is covered later in this section.)

Table 6. Properties of the static RegExp object.

When a regular expression object is evaluated, the static RegExp object is also filled with some properties that can be handy in your script. When a regular expression executes, as shown in the following example, the properties listed in Table 6 become available from the static RegExp object:

```
message = "'I can see the farmer seeding seeds.'," +
          "she said. 'I See,' replied pig./";

var exp = /\bsee\b/gi;
arrayMessage = exp.exec(message);
```

The RegExp object now contains some good information about the evaluated expression.

Figure 6 shows the values that are the static RegExp object returns.

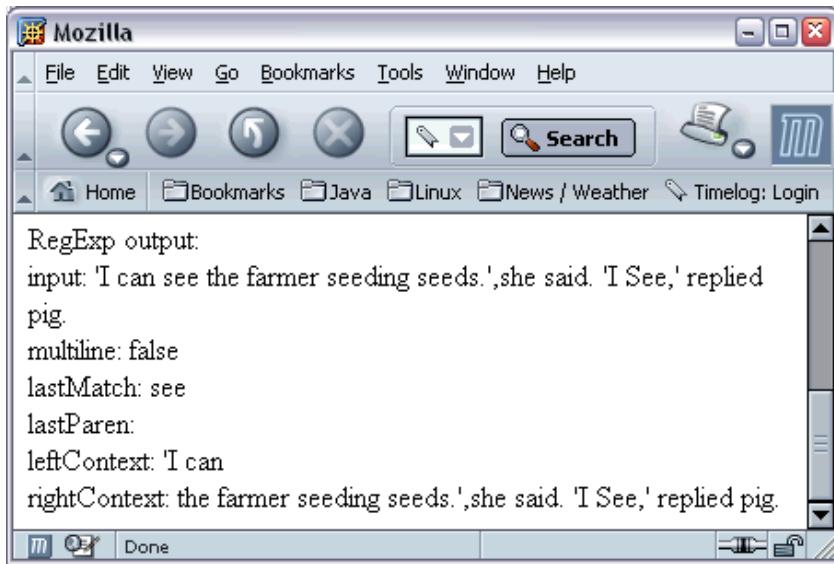


Figure 6. Mozilla RegExp object return values.

As you might expect, the input property returns the string that was evaluated. The multiline property is false in this case, because the string consists of a single-line sentence. The lastMatch property indicates the last substring matched, which was *see*; to the left of that match was the leftContext property *I can*; and to the right of it was the rightContext property *the farmer seeding seeds', she said. 'I see,' replied pig*.

In addition, the arrayMessage variable is assigned two new properties, index and input. In this case, arrayMessage[0] would be *see*, the index would be 0, and input would be the original string that was evaluated. When you set an array this way, only the first match that the expression encounters is added to

arrayMessage, which is why the index is 0. To find all of the matches, you would have to set global to true, and continue iterating .exec(). When the regular expression object is null, you have found all the matches. For intermediate matching, it is much easier to use string.match(expression).

See  
*RegExpFun.html*

When you test your regular expressions, you can save time by using the provided script located in the *RegExpFun.html* file.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
  "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
<body>

<script language="JavaScript">
<!--
function getTimes() {
    return this * this;
}

Number.prototype.timesMe = getTimes;

ten = 10;
ten.timesMe

function expFun(exp, str) {
    var re = eval(exp);
    var ar = str.match(re);
    var arexec = re.exec(str);
    var res = "Results:\n\n";
    var regexprops = "input,multiline,lastMatch,lastParen"
        + ",leftContext,rightContext";
    regexprops = regexprops.split(",");
    var reprops = "source,global,ignoreCase,lastIndex";
    reprops = reprops.split(",");
}
```

```
if (ar) {
    for (l = 0; l < regxprops.length; l++) {
        res += "RegExp." + regxprops[l] + ": " +
            eval("RegExp." + regxprops[l]) +
            "\n";
    }

    res += "Backreferencing:\n";
    for (br = 0; br < 9; br++) {
        res += "RegExp.$" + (br + 1);
        res += ": " +
            eval("RegExp.$" + (br + 1))
            + "\n";
    }

    res += "\n";
    res += "Regular Expression Object:\n";

    for (r = 0; r < reprops.length; r++) {
        res += "re." + reprops[r] + ": " +
            eval("re." + reprops[r]) +
            "\n";
    }

    res += "\n";
}

res += "Regular Expression Object Array:\n";

res += "arexec[0]: " + arexec[0] + "\n";
res += "arexec.index: " + arexec.index + "\n";
res += "arexec.input: " + arexec.input + "\n\n";

res += "Matches from string.match():\n"

for (m = 0; m < ar.length; m++) {
    res += "array[" + m + "]: " + ar[m] + "\n";
}
```

```
        return res;

    } else {
        return "No matches";
    }
}

//-->
</script>

<form name="form1">
    RegExp: <input type="text" name="expression"
        value="/hello/gi" size="50"/><br/>
    String: <input type="text" name="string"
        value="hello, hellohello, " +
        "how are you today \n hello!"/><br/>
    <input type="button" name="button"
        value="Express!"
        onclick="document.form1.result.
            value = expFun(document.form1.expression.value,
            document.form1.string.value);" /><br/>

    <textarea cols="40" rows="25"
        name="result">Results</textarea>

</form>

</body>
</html>
```

# Summary

- Strings can have properties and methods that are prototyped, which extends the basic functionality of the String object.
- A string's charAt method extracts the character located at a particular index in the string.
- The charCodeAt method of an index in a string results in the ASCII character code of that character.
- Create arrays of particular parts of a string by using the match() method and providing it with a regular expression.
- Replace certain patterns in a string by using the String.replace(RegExp, *string*) method.
- A string can be split into an array by providing a delimiter or regular expression to the split() method.
- Months and days are returned from the Date object starting at 0, which is January in months, or Sunday in days.
- Create a date for the current time by just calling new Date().
- Several strings can be inserted in the Date object's constructor, which enable you to create a date for a specific point in time; for example: new Date(yy, mm, dd).
- JavaScript's global setInterval() method is used for creating routines that must fire at a defined interval.
- The setTimeout function is used for routines, such as a stopwatch, that need to wait until a specific time to fire, or to fire continuously until a specific time is reached.
- The Math object provides some basic math methods and constants to make doing math routines and algorithms easier.
- Use methods as parameters to other methods, as in this example, which generates a random number between one and ten:  
`Math.floor(Math.random() * 10).`
- A regular expression can be used to match specific sequences and patterns in strings.
- Adding g after a regular expression ensures that the expression searches completely and does not simply stop after the first match.

# Questions

1. If you have created a string variable called *address* and defined a string object prototype method with the statement `String.prototype.alertstring = alertFunction`, how would you use this new method with your string?
2. Which method do you use to determine what character is located at a particular index in a string?
3. What are the two methods you can use to split a string by using its `split` method?
4. The month January returns from the Date object as what number?
5. To create a Date object that equals the current time, what syntax do you use?
6. If you want to launch a pop-up window 3 ½ minutes after your page loads, do you use `setInterval` or `setTimeout`?
7. True/False: You must create a new `Number()` object to access a prototyped `Number` method.
8. How do you return a random number between 0 and 10?
9. Define a simple regular expression that finds all occurrences of the word *butterfly* in a sentence, regardless of case.

# Answers

1. If you have created a string variable called *address* and defined a string object prototype method with the statement `String.prototype.alertstring = alertFunction`, how would you use this new method with your string?  
**address.alertstring**
2. Which method do you use to determine what character is located at a particular index in a string?  
**charAt(index)**
3. What are the two methods you can use to split a string by using it's split method?  
**( delimiter, int ) or ( RegExp )**
4. The month January returns from the Date object as what number?  
**0**
5. To create a Date object that equals the current time, what syntax do you use?  
**var adate = new Date()**
6. If you want to launch a pop-up window 3 ½ minutes after your page loads, do you use setInterval or setTimeout?  
**setTimeout**
7. True/False: You must create a new Number() object to access a prototyped Number method.  
**False; you can also do this with a number variable.**
8. How do you return a random number between 0 and 10?  
**Math.floor(Math.random() \* 10)**
9. Define a simple regular expression that finds all occurrences of the word *butterfly* in a sentence, regardless of case.  
**/butterfly/gi**

# Lab 6: Built-In Objects

**TIP:** Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You will find all the completed code in **ValidateMe.html** and **ValidateMe.js**, in the same directory as the sample project. To avoid typing the code, use the exercise specific .js files **ValidateMe\_ex1.js**, for exercise one, and **ValidateMe\_ex2.js** and **ValidateMe\_ex3.js** respectively. Once you have completed each part, rename it **ValidateMe.js** to test it.

# Lab 6 Overview

In this lab you will learn to properly access various form elements, and how to validate values.

To complete this lab, you will need to work through three exercises:

- Terminal: Routing and Setup
- Defining Validations
- Display Results

Each exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

# Terminal: Routing and Setup

## Objective

In this exercise, you will use a decision construct to direct different form elements to the appropriate validation function.

## Things to Consider

- The type property for a select object returns as ‘select-one’.
- Text objects and Textarea objects can be validated similarly.
- You cannot return the length property of a radio group accurately from the form.elements[] array. The reason is that when you call length on a single radio object, it always returns null.
- The onSubmit() event handler of the form accepts a Boolean value indicating whether or not to process the submit request.

## Step-by-Step Instructions

1. You will be using the form found in the **ValidateMe.html** file. It would be helpful to glance through this file before you start the exercise, and take note of a few things within the file:
  - The file will look for the external .js file, **ValidateMe.js**.
  - The onsubmit event of **form1** calls the function **validateForm()**, which serves as the main function of this application.
  - By design, form objects that have names beginning with underscore are required fields.
2. Open **ValidateMe.js** and **ValidateMe.html**. In **ValidateMe.js**, create the main function, **validateForm()**. Have the function accept a parameter called **frm**, which will be used to accept a form object.

```
function validateForm(frm) {  
}
```

3. In the first few lines of the validateForm() function, define the variables **submitform = true**, **reqChar = “\_”**, and **currElement**.

**NOTE** The **submitform** variable will ultimately determine whether the form is submitted or not. The **reqChar** variable represents the first character in the name of a required field on the form. The script, when complete, will know to look for this character to make sure that that current object has a value. Finally, the **currElement** variable represents the current element in any iteration of the **forms.elements[]** array. This is important to note, because when the **ElementLoop** is stopped because of a validation error, the **currElement** will hold the information about the object that was incorrect.

```
function validateForm(frm) {  
    var submitform = true;  
    var reqChar = "_";  
    var currElement;
```

4. Under the variable declarations, create a label called **ElementLoop** and under that, create a for loop block. Iterate the loop for the length of **frm** with the condition: **fo = 0; fo < frm.elements.length; fo++**.

```
    var submitform = true;  
    var reqChar = "_";  
    var currElement;  
ElementLoop:  
    for (fo = 0; fo < frm.elements.length; fo++) {  
  
    }
```

5. Within the loop block, create an if block. The purpose of the if block will be to catch only those elements that have the **reqChar** as the character at the first position in the string (**charAt(0)**), and to ignore any undefined elements (like the fieldset object).

```

ElementLoop:
for (fo = 0; fo < frm.elements.length; fo++) {
    if (frm.elements[fo].name != undefined &&
        frm.elements[fo].name.charAt(0) == reqChar) {
        //Then this is an element to be validated.
    }
}

```

6. The first statement in the if block must set **currentElement** to reference the currently selected form element in the iteration.

```

if (frm.elements[fo].name != undefined &&
    frm.elements[fo].name.charAt(0) == reqChar) {
    currElement = frm.elements[fo];
}

```

7. On the next line, create a switch block with four cases, and no default block. The switch condition will be (**currElement.type**). The cases represent the element types that you want to validate. For this exercise they will be as follows: **case “select-one”**, **case “radio”**, **case “text”**, and **case “textarea”**. Be sure to add a **break;** statement to each case.

```

if (frm.elements[fo].name != undefined &&
    frm.elements[fo].name.charAt(0) == reqChar) {
    currElement = frm.elements[fo];

    switch(currElement.type) {
        case "select-one":
            break;
        case "radio":
            break;
        case "text":
            break;
        case "textarea":
            break;
    }
}

```

8. Right before the end of the main elements loop, add another if block with the statement **if (!submitform) { break ElementLoop; }**. This ensures that if an element does not evaluate to true, then the loop will stop iterating.

```
        case "textarea":  
            break;  
        }  
    }  
  
    if (!submitform) {  
        break ElementLoop;  
    }  
}
```

9. Underneath the main function, define the validation utility functions. There will be a validation function for each type of object caught by the switch statement in **validateForm**. Each one will accept a **frmobj**, which is also the **currElement** of the form. Name the functions **valSelect()**, **valRadio()**, and **valText()**.

```
function valSelect(frmobj) {  
}  
  
function valRadio(frmobj) {  
}  
  
function valText(frmobj) {  
}
```

10. One thing that each function has in common is a Boolean variable called **valid**. Start by defining **valid** as false; each function will eventually return the **valid** variable.

```
function valSelect(frmobj) {  
    var valid = false;  
  
    return valid;  
}  
  
function valRadio(frmobj) {  
    var valid = false;  
  
    return valid;  
}  
  
function valText(frmobj) {  
    var valid = false;  
  
    return valid;  
}
```

11. In the validateForm() function, each case is going to be nearly identical. Each one will define **submitform** using its corresponding utility function. For example, case “select-one” will define **submitform** by assigning it the return value of the function **varSelect(currElement)**.

**NOTE** The one exception to the rule is case “radio”. When the element in question is a radio button, **currElement** actually references an individual radio button object. The default value of an individual radio button’s length is null. In this application, you just want to make sure one of several radio buttons is selected, so you must access the length attribute of the whole radio group. This is available by establishing a direct reference to the radio group name on the form (**document.formname.radiogroupname**), which coincidentally is the same as each radio button’s name.

```
switch(currElement.type) {  
    case "select-one":  
        submitform = valSelect(currElement);  
        break;  
    case "radio":  
        submitform = valRadio(eval("document."  
            + frm.name + "." +  
            currElement.name));  
        break;  
    case "text":  
        submitform = valText(currElement);  
        break;  
    case "textarea":  
        submitform = valText(currElement);  
        break;  
}
```

12. At the very end of the validateForm() function, the code determines whether to submit or not. So beneath the Element Loop, start an if/else statement, with the condition **submitform**. If submitform is true, return true from validateForm() back to the **onsubmit** event handler that called it. If submitform is false, return false.

```
if (submitform) {  
    return true;  
} else {  
    return false;  
}
```

13. If the form validated as true, the application will generate an alert message: “**The form is complete!**”. If it is false, it will generate an alert message saying “**The selected field was not completed.**”, and then set focus to the element that is not valid. Because the ElementLoop breaks when a problem occurs, currElement will be the violating element.

```

if (submitform) {
    alert("The form is complete!");
    return true;
} else {
    alert("The selected field was not completed.");
    currElement.focus();
    return false;
}

```

14. Now you can test the exercise. The **submitform** variable will be false each time, because the validation utility functions have not yet been defined. Just make sure that there are no errors, and continue on to the next exercise. The result should look something like Figure 7.

**TIP:** A form button called **shortcut** and its corresponding function have been added to the exercise files, so you can instantly fill in the form when shortcut is clicked to save some keystrokes.

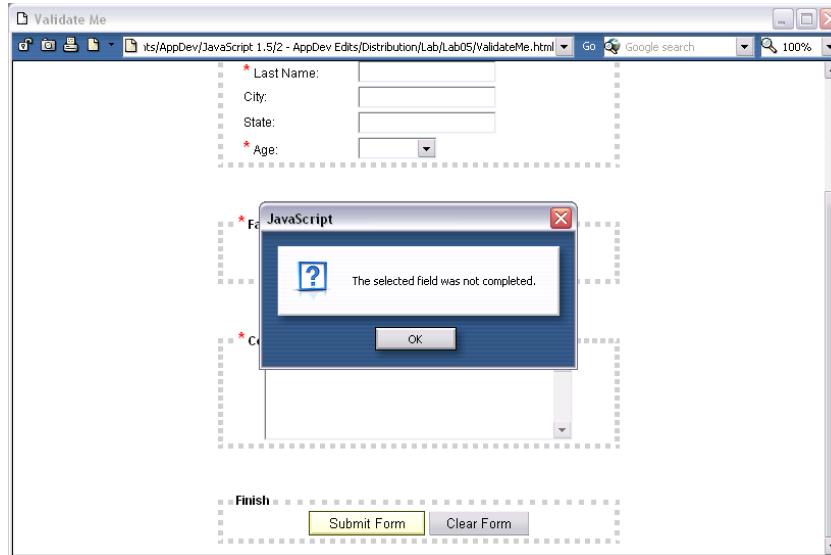


Figure 7. The result of this exercise.

# Defining Validations

## Objective

In this exercise, you will set up the actual validation for the text, textarea, radio, and select objects.

## Things to Consider

- For this example the select object's validation rule is that the current selected option must have a value, and may not contain an empty string. The default option, [Select], has an empty string for its value, and would therefore be considered invalid if it were submitted with the form.
- For the radio group, none of the radio buttons are checked by default, but one radio button must be checked by the user to be considered a valid submission.
- The default value for the text and textarea objects is an empty string, but these elements must contain some character data in order to be considered valid entries.

## Step-by-Step Instructions

1. You will start by defining the text/textarea validation function, **valText()** from the previous exercise. Create an if block just after the declaration of the valid variable. The expression for the if statement will be **(frmobj.value.length > 0)**, to test whether the length of the entry is greater than zero; if so, the valid variable must be set to true with the statement **valid = true;**

```
function valText(frmobj) {  
    var valid = false;  
    if (frmobj.value.length > 0) {  
        valid = true;  
    }  
    return valid;  
}
```

2. Next, define the select validation in the function **valSelect()**. From the chapter you should remember that a select object has a **selectedIndex** property, which will return the selected object's index position in the options array. Add an if block between the **valid** variable definition and the **return valid;** statement that has the condition **(frmobj[frmobj.selectedIndex].value.length > 0)**. This checks to make sure that the selected Index has value.

**NOTE** In Mozilla, the default value or the **[Select]** instruction-option must have a value of “” (an empty string) defined, or it will return the selected option’s label (i.e., **[Select]**) as the value.

```
function valSelect(frmobj) {
    var valid = false;
    if (frmobj[frmobj.selectedIndex].value.length > 0) {
        valid = true;
    }
    return valid;
}
```

3. Define the **varRadio(frmobj)** function, which will loop through the objects in the radio group parameter that is passed in. Create a for loop block with the condition **(ri = 0; ri < frmobj.length; ri++)**.

```
function valRadio(frmobj) {
    var valid = false;
    for (ri = 0; ri < frmobj.length; ri++) {
        if (frmobj[ri].checked) {

        }
    }
    return valid;
}
```

4. Within the loop block that you created in Step 3, create an if block with the condition **(frmobj[ri].checked)**. If the current radio object in the radiogroup is checked, set the variable valid to true, using the statement **valid = true;** On the next line, add a **break;** statement to ensure that the code stops executing the loop once a checked radio object is found.

```
function valRadio(frmobj) {  
    var valid = false;  
    for (ri = 0; ri < frmobj.length; ri++) {  
        if (frmobj[ri].checked) {  
            valid = true;  
            break;  
        }  
    }  
    return valid;  
}
```

5. Now it is time to test the script. Submit the form once with only one type of form object selected (and the other types not selected) so that you can check each validation function. If a required element is not selected or filled in, it should receive focus after you click the **OK** button in the alert message dialog box, as shown in Figure 8.

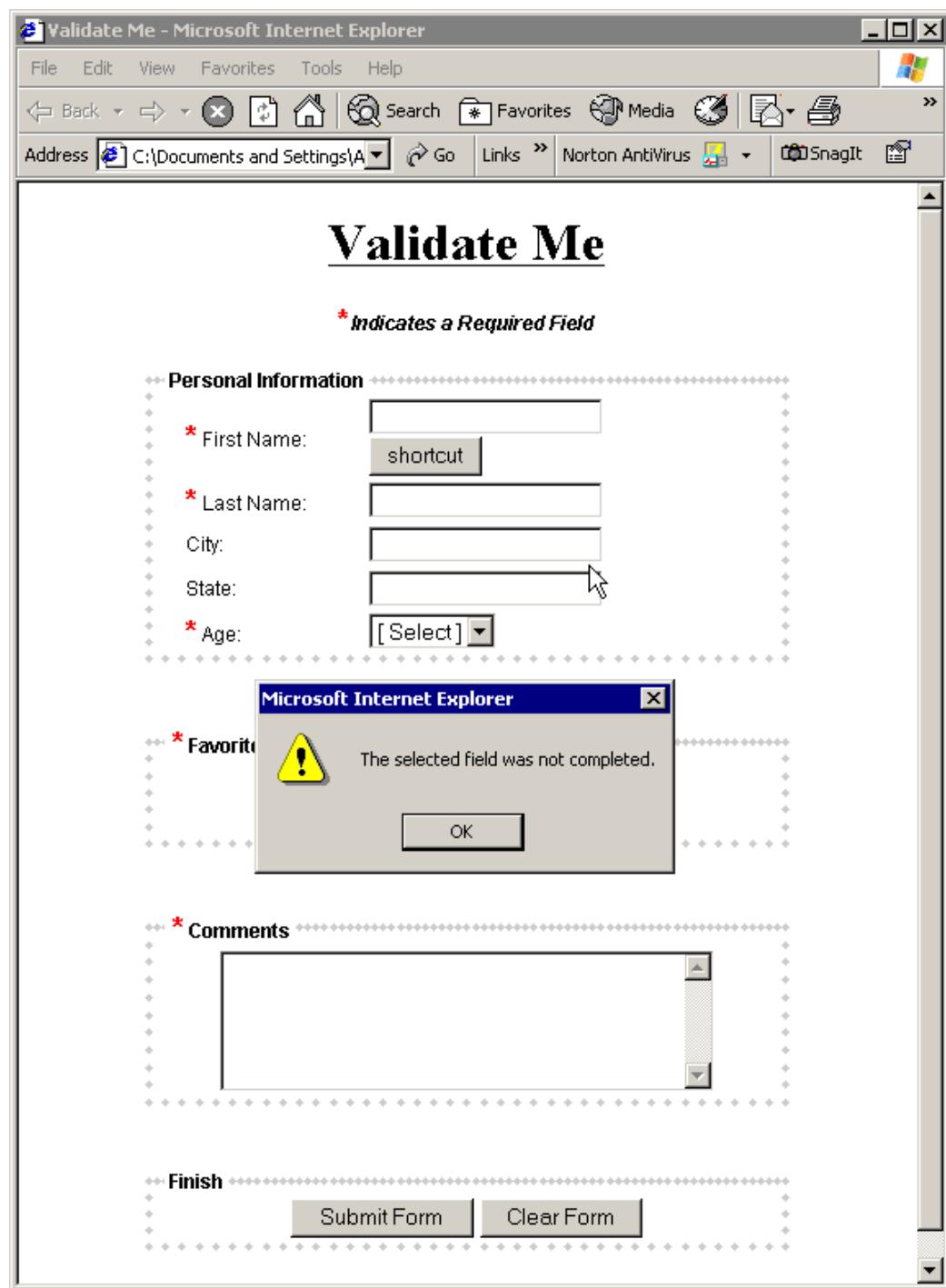


Figure 8. The alert message box in ValidateMe.html.

# Display Results

## Objective

In this exercise, you will parse through and display the values of the form after a successful submission by parsing through the query string for the page.

## Things to Consider

- The location object's search method returns everything after the base URL on the address bar (the question mark and everything after it).
- The form only passes a query string to the “action” page when a **GET** method is used.

## Step-by-Step Instructions

1. Define a global variable at the bottom of the **ValidateMe.js** file called **submitmsg**, and set it to reference an empty string.

```
var submitmsg = "";
```

2. On the next line, define an if block with the condition: **(window.document.location.search.length > 0)**. This tests to see if there are any parameters in the querystring, which means that the form has been submitted.

```
if (window.document.location.search.length > 0) {  
}
```

3. Next, the application will build XHTML output into the **submitmsg** variable, to be displayed on the page.

**NOTE** There is already a place in the **ValidateMe.html** file that checks **submitmsg**, and writes the contents if it has value:

```
<!-- Display Submit Results -->

<script language="JavaScript" type="text/javascript">
    //submitmsg is declared in 'ValidateMe.js'

    if (submitmsg) {
        document.write(submitmsg);
    }
</script>
```

4. On the first line of the if statement from Step 3, create a new variable called **qs** and assign it the value of **window.document.location.search**, which is the query string with the question mark at the beginning. On the next line, create an array of name-value pairs called **aQueryString**. Define it by getting the name/value pairs from the query string by first removing the question mark, then splitting the querystring at the “**&**” character.

**TIP:** So you can better see what you are working with, the query string will look like this:  
**?\_fname=111&\_lname=112&city=113&state=114&\_age=1020&\_favsite=**  
**Yahoo&\_comments=115&submit=Submit+Form**

```
var qs = window.document.location.search;
var aQueryString = (qs.substr(1,
    qs.length)).split("&");
```

The array now equals a series of name/value strings that look something like: “**\_fname=111**”. To extract the name and value, the string will have to be split again into an array using the **=** character.

5. Next you will start creating the **submitmsg** to display in **ValidateMe.html**. Start with the formatting, which you can copy from the following code snippet:

```
submitmsg = "<fieldset class=\"fieldset\"><legend  
class=\"text\"><b>";  
submitmsg += "Submission Results</b></legend><div  
align=\"center\">";  
submitmsg += "<font class=\"text\" color=\"green\">";  
  
// INSERT NAME / VALUES HERE  
  
submitmsg += "</font></div></fieldset>";
```

6. Add a for loop where the comment **// INSERT NAME / VALUES HERE** is located, with the condition (**al = 0; al < aQueryString.length; al++**). This loop will iterate through the name-value pairs that were extracted into the **aQueryString** array. For the first statement in the loop, split up the name and value of the current pair into another temporary array, using the statement **a = aQueryString[al].split("=")**;

```
for (al = 0; al < aQueryString.length; al++) {  
    var a = aQueryString[al].split("=");  
  
}
```

7. For the second statement in the loop, output the name and value of the current index in the array in XHTML. Remember, **a[0]** references the name and **a[1]** references the value:

```
for (al = 0; al < aQueryString.length; al++) {  
    var a = aQueryString[al].split("=");  
    submitmsg += ("<b>" + a[0] + "</b> is equal to  
    <b>" + a[1] + "</b><br/>");  
}
```

8. Test the completed project by launching **ValidateMe.html**. Once you have completed the form and submitted it, all the values should be shown on the page, with the complete form resembling the one in Figure 9.

The screenshot shows a web browser window titled "Validate Me". The address bar contains the URL "8&state=114&\_age=1020&\_fbsite=Yahoo&\_comments=115&submit=Submit+Form". The page content is divided into sections: "Submission Results" at the top, followed by "Personal Information" and a "Favorite Internet Site" section. The "Submission Results" section lists validation messages for various form fields. The "Personal Information" section contains input fields for First Name, Last Name, City, State, and Age. The "Favorite Internet Site" section has a single input field. A note at the bottom indicates that the "fbsite" field is required.

**Submission Results**

- \_fname is equal to 111
- \_lname is equal to 112
- city is equal to 113
- state is equal to 114
- \_age is equal to 1020
- \_fbsite is equal to Yahoo
- \_comments is equal to 115
- submit is equal to Submit+Form

**Validate Me**

\* Indicates a Required Field

**Personal Information**

* First Name:	<input type="text"/>	shortcut
* Last Name:	<input type="text"/>	
City:	<input type="text"/>	
State:	<input type="text"/>	
* Age:	<input type="button" value="Select"/>	[ Select ]

\* Favorite Internet Site

Figure 9. The final page after it has been completed successfully.

*Lab 6:  
Built-In Objects*

---

Feb 19 2008 3:29PM Dao Dung dungdq@edt.com.vn  
For produ6-54valuation only– not for distribution or commercial use. **JavaScript 1.5**

Copyright © 2003 by Application Developers Training Company  
All rights reserved. Reproduction is strictly prohibited.



# Windows and Frames

## Objectives

- Learn how to spawn a new window using JavaScript.
- Create content for windows spawned through JavaScript.
- Learn how to launch functions in spawned windows.
- See how to pass values to spawned windows.
- Discover how to modify attributes.
- Understand how to use modal dialog boxes in Internet Explorer 5+.
- Discover how to create a frameset.
- Understand the relationship between parent and child frames.
- Learn how to use inline iframes.

# The Window Object

In order to make effective use of windows and frames, you need to have a good understanding of the window object. The window object, which is at the top of the document object model hierarchy, is the outermost container of all document-related objects that you can script. It encompasses all of the properties and behavior of the browser window itself, from the content area of the browser, to the scrollbars, toolbars, menu bar, and status bar (also known as chrome), and even the dimensions of the window.

**NOTE** Not all browsers provide you with full scripted control of the chrome of the main browser window, but you may spawn a new window with exactly the chrome elements you desire.

You can reference the properties and methods of the window object in several ways. The most common way is to include the window object in the reference:

```
window.propertyName  
window.methodName ( [params] )
```

When your script references the window that contains the document, you may also use the synonym self:

```
self.propertyName  
self.methodName ( [params] )
```

Either of these object references may be used interchangeably, but the use of the self keyword is preferable because it specifies more clearly that the code refers to the current window that holds the script's document. This makes complicated scripts, involving multiple frames, much easier to read.

When the script references properties and methods of the current window, you may also omit the object reference altogether:

```
propertyName  
methodName ( [params] )
```

## Spawning a Window

When the browser displays a page for the user, it always opens the main browser window. In addition, you can use script within a page's document to open any number of sub-windows. The syntax of the method to spawn a new sub-window, also called a child window, is:

```
window.open("URL", "windowName" [, "windowFeatures"])
```

The `window.open()` method takes up to three string parameters, which define the window properties: the URL for the document to be loaded, a name for TARGET references in HTML tags, and an optional parameter that can be used to specify the physical appearance of the window. The following example illustrates the usage of `window.open()` to spawn a new browser window whose document resides at [www.appdev.com](http://www.appdev.com):

```
var newWindow = window.open("http://www.appdev.com",
    "appDev")
```

The `window.open()` command in the example above omitted the third parameter, which defines the dimensions and chrome of the new window. When you omit this parameter, the new child window will retain the look and feel of the parent window that spawned it. If you do specify a value for the third parameter, then the attributes that you define explicitly apply only to the child window. Table 1 lists the most commonly used attributes for the parameter, their expected value type, and the browsers that support the attribute.

Attribute	Supported Browsers	Expected Value Type	Description
channelmode	IE4+	Boolean	Enables Theater mode
dependent	NN4+	Boolean	The spawned window closes if the originating window closes
directories	NN2+, IE3+	Boolean	
fullscreen	IE4+	Boolean	No title bar or menus are displayed
height	NN2+, IE3+	Integer	Height of content region in pixels
left	IE4+	Integer	Horizontal position of top-left corner
location	NN2+, IE3+	Boolean	Displays the address bar
menubar	NN2+, IE3+	Boolean	Displays the menu bar at the top of the window
resizable	NN2+, IE3+	Boolean	Allows resizing of window by dragging
scrollbars	NN2+, IE3+	Boolean	Displays scrollbars if the document is larger than the window
status	NN2+, IE3+	Boolean	Displays the status bar at the bottom of the window
title	IE5	Boolean	Displays the title bar
toolbar	NN2+, IE3+	Boolean	Displays the toolbar with navigation buttons
top	IE4+	Integer	Horizontal position of top-left corner on screen
width	NN2+, IE3+	Integer	Width of content region in pixels

Table 1. Useful attributes that are controllable with the `window.open()` method.

## Referencing the New Window

It is important to note that the statement in the previous example was an assignment statement. The `window.open()` method, in addition to creating a new window object and setting its initial properties, returns a reference to that window object and assigns it to the variable:

```
var newWindow = window.open("http://www.appdev.com",
    "appDev") // "newWindow" references the child window
```

Now you can use that variable as a reference to access the methods and properties of the new child window and create content for it.

**TIP:** If no other statements within a document need to reference the new window, you don't need to assign the return value to a variable.

For example, if you want to close your newly created window via a script contained in either the parent window or the child window, you could simply call the `close()` method of the `newWindow` variable, as follows:

```
newWindow.close();
```

**WARNING!** It is a good practice to reference the specific window you want to close in your script. If you were to call `window.close()`, `self.close()`, or just `close()` from a script contained in the parent window, then you would close the main window itself.

## Creating Content in the New Window

In most cases, the content for a spawned window resides in a preexisting HTML file, which you define as the first parameter in your call to `window.open()`. This is fine for pages with static content, but often you will want dynamic control over the content of your new window. In this case, you can use `document.write()` statements in a script that is referenced by the child window to create the dynamic content.

After a page loads, the browser's output stream closes automatically. When the output stream closes, any content in the stream displays in the browser. Any

further calls to document.write() once a stream is closed will open a new stream and erase the current page. Once you open a new stream, you should be sure to always close it through a call to the document.close() method, or your content may not display correctly. The reason is that any call to document.open() on an already open stream will append the new content to the old. Any HTML tags that are intended to overwrite those of the old document will not take effect, and any body text in the new document will likely be appended to the old.

*See*  
***ChildContent.html*** The following example creates a page that spawns a blank child window when it loads. The parent window has two buttons that, when pressed, dynamically build different page content in the form of a text string and pass it through a call to the child window's document.write() method to change the display.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml
xml:lang="en" lang="en">
<head>
<title>Dynamic Content for Sub-windows</title>

<script language="JavaScript" type="text/javascript">
var newWindow;

function spawnWindow() {
    //opens a small window with no content
    newWindow = window.open("", "newWindow",
                           "height=300, width=300");
}

function writeHello() {
    //if window doesn't exist, or has closed, open it
    if (newWindow.closed) {
        spawnWindow();
    }
    newWindow.focus();
    var pageContent = "<html><head>";
    pageContent += "<title>New Window</title></head>";
    pageContent += "<body bgcolor='blue'" +

```

```
        "text='black'>";
    pageContent += "<H1>Hello World!</H1>";
    pageContent += "</body><html>";
    //write the content
    newWindow.document.write(pageContent);
    //close the layout stream
    newWindow.document.close();
}

function writeGoodbye() {
    //if window doesn't exist, or has closed, open it
    if (newWindow.closed) {
        spawnWindow();
    }
    newWindow.focus();
    var pageContent = "<html><head>";
    pageContent += "<title>New Window</title></head>";
    pageContent = "<body bgcolor='black' text='red'>";
    pageContent += "<H1>Goodbye World!</H1>";
    pageContent += "</body><html>";
    //write the content
    newWindow.document.write(pageContent);
    //close the layout stream
    newWindow.document.close();

}
</script>

</head>
<body onload="spawnWindow()">
    <input type="button" value="Hello World"
           onClick="writeHello()">
    <input type="button" value="Goodbye World"
           onClick="writeGoodbye()">
</body>
</html>
```

## Launching Functions and Passing Values

In the previous example, you saw how to modify the content of a spawned child window by calling its `document.write()` method from the parent window. In much the same way, you are able to define custom functions in the child window and call them from the parent window.

*See  
[ChildFunctions\\_Main.html](#)*

The following example uses two pages. The first, called `ChildFunctions_Main.html`, is the parent page and opens a new child window in its `onLoad` event. The child window, defined in the code listing `ChildFunctions_Sub.html`, has a single button whose `onClick` event calls a function defined within a script of the new window.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
  "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
  <head>
    <title>Main Window</title>

    <script language="JavaScript" type="text/javascript">
      var newWindow;
    </script>

  </head>
  <body onload="newWindow = window.open(
    'ChildFunctions_Sub.html', 'newWindow',
    'height=300,width=300,status')">

    <input type="button"
          value="Change Child Background Colors"
          onClick="newWindow.changeBackgroundColor()" >
  </body>
</html>
```

See  
*ChildFunctions\_Sub.html*

The code for the child window simply defines a function that, when executed, changes the background color of the document displayed in the child window.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="en" lang="en">
<head>
<title>Subwindow</title>

<script language="JavaScript" type="text/javascript">
function changeBackgroundColor() {
    focus();
    switch (document.bgColor) {
        //Check if background is white.
        case "#ffffff":
            //Change background to blue.
            document.bgColor="#0000ff";
            break;
        //Check if background is blue.
        case "#0000ff":
            //Change background to red.
            document.bgColor="#ff0000";
            break;
        //Check if background is red.
        case "#ff0000":
            //Change background to white.
            document.bgColor="#ffffff";
            break;
    }
}
</script>
</head><body></body></html>
```

In this example, when you click the button on the main page, it calls and executes the function defined within the child window. If the function on the child page accepts parameters, you could also pass values derived on the main page to the child via those function call parameters.

In addition, you can alter the value of a global variable within the child page by using the parent page's variable reference to the child page. The following example modifies the pages from the previous example slightly. The child page declares a variable called *counter* and initializes it to 0.

```
<script language="JavaScript" type="text/javascript">
    var counter = 0;

    function changeBackgroundColor() {
        ... // code to change background color
    }
}
```

The following code creates a new function on the parent page that increments the counter variable declared on the child page and writes it to the status bar of the child window. Normally, it is considered bad practice to write any text to the status bar unless it is in response to an event with a temporary effect. However, it will simplify the example to overlook this rule of thumb just this once.

```
<script language="JavaScript" type="text/javascript">
    var newWindow;

    function writeChildCounter() {
        newWindow.focus();
        newWindow.counter += 1;
        newWindow.status = "Counter:" + newWindow.counter;
    }
</script>
```

Finally, the code adds a new button to the parent form that calls the *writeChildCounter()* function in its *onClick* event:

```
<input type="button"
       value="Change Child Background Colors"
       onClick="newWindow.changeBackgroundColor () ">

<input type="button"
       value="Increment Counter on Child"
       onClick="writeChildCounter () ">

</body>
```

When you run the code with these modifications, you'll notice that each time you click the new button, an integer value on the status bar is incremented. Examining the code reveals that the `writeChildCounter()` function increments the child's global variable *counter* before it updates the status bar of the child form.

Just as a parent window can access the variables, functions, and objects of a child window that it creates, a new child window can access the elements of its parent. This is possible because every window has an `opener` property that references the window or frame that created it. Just as you've seen with variable references to spawned child windows, you can use the `opener` property in a statement that references the parent window.

**NOTE** For the main browser window, the value of *opener* is always null.

Building on the previous example, the spawned window in the following code notifies its parent every time the child window's background color changes.

```
function changeBackgroundColor() {
    focus();
    switch (document.bgColor) {
        //Check if background is white.
        case "#ffffff":
            //Change background to blue.
            document.bgColor="#0000ff";
            break;
        //Check if background is blue.
        case "#0000ff":
            //Change background to red.
            document.bgColor="#ff0000";
            break;
        //Check if background is red.
        case "#ff0000":
            //Change background to white.
            document.bgColor="#ffffff";
            break;
    }
    opener.alert("The child window's background " +
        "color is now " + document.bgColor);
}
```

## Modal and Modeless Dialog Boxes in Internet Explorer 5+

A feature specific to Internet Explorer 5+ is the ability to open a modal dialog box. A modal dialog box stays on top of the main browser window and maintains focus, thereby preventing the user from accessing the main browser window behind it. In addition, script execution in the main window halts from the time that the modal dialog statement executes until the user closes the modal dialog box. When the modal dialog box closes, the script that was running in the main window resumes execution where it stopped.

The syntax to create a modal dialog box is:

```
showModalDialog( "URL" [, arguments] [,features]);
```

The parameters used for the showModalDialog() method are much the same as those for the window.open() method. The first parameter is the URL of the page or image to open in the dialog box. The second parameter is optional and allows you to pass data to the modal dialog box. The third parameter is also optional, and allows you to format the look of the dialog box.

The method can return a value, but it does not automatically do so. If you set the window.returnValue property of the dialog box before the dialog box closes, the showModalDialog() method will return whatever value is assigned to the returnValue property upon completion. The value that a modal dialog box returns can be of any valid JavaScript data type.

There is also a variation of the modal dialog box, called a modeless dialog box. Like a modal dialog box, the modeless dialog box stays on top of the main browser window, but allows the user to access the main window. It also does not halt scripts that are running in the main window. The syntax to create a modeless dialog box is:

```
showModelessDialog("URL" [, arguments] [, features]);
```

Because a modeless dialog box does not prevent the change of data on the main form, returning data from one can be complicated. For this reason, the modeless dialog box accepts a reference to a variable or function in the main browser window that will accept the data.

**NOTE** Neither a modal nor a modeless dialog box can access the originating window through its opener property. In both cases, the opener property is undefined.

## Creating a Frameset

Now that you know how to deal with multiple windows and their various interactions, you are ready to learn about frames. With a traditional Web page, the browser opens a single window object and displays a single document within it. However, since frames are designed to display multiple pages at once, things become a bit more complicated. With frames, the browser opens a page called a frameset, which is essentially a document area container that appears within the browser window. The frameset defines multiple boundaries or frames that can each display a separate document.

From a practical standpoint, the frameset itself functions like a window object within the main browser window. Although it has all the properties and methods of a standard window object, such as a document object, the frameset itself can't contain most HTML objects, such as forms or controls. Its primary purpose is to maintain the relationships between the frames within it, and those relationships are defined in the frameset's document.

Of course, the frames themselves are window objects with their own documents. Figure 1 further illustrates the typical document hierarchy of a two-frame frameset.

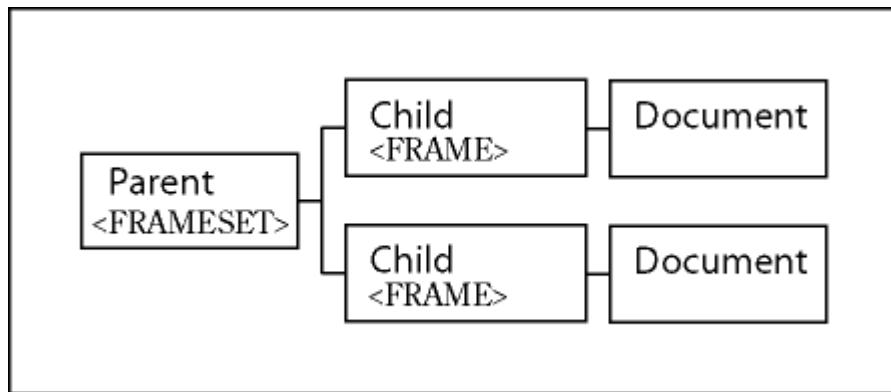


Figure 1. The document hierarchy of a window with two frames.

See  
[SimpleFrameset\\_Cols.html](#)

The following example illustrates how to define frames within a framesetting document:

```
<html>
<head><title>Simple Frameset with Columns</title></head>
<frameset cols="50%,50%">
    <frame name="leftFrame" src="http://www.appdev.com">
    <frame name="rightFrame" src="http://www.google.com">
</frameset>
</html>
```

See  
[SimpleFrameset\\_Rowshtml](#)

The preceding example creates a frameset with two frames side by side, each using half of the browser window. Each frame must point to an HTML source, which defines what appears in the content area of the frame. It is also advisable to assign a name to each frame with the name attribute, to make it easy to reference each one in your scripts. You can define additional frames, each occupying whatever percentage of the browser page you desire. In addition, you are not restricted to arranging frames in columns. The following example creates three frames arranged in rows, each with differing sizes:

```
<html>
<head><title>Simple Frameset with Rows</title></head>
<frameset rows="10%,30%,60%">
    <frame name="topFrame" src="http://www.appdev.com">
    <frame name="midFrame" src="http://www.google.com">
    <frame name="botFrame" src="http://www.slashdot.org">
</frameset>
</html>
```

## Parents and Children

The framesetting document's primary purpose is to maintain the relationships among its frames. It is easiest to conceptualize the relationship between the window of a framesetting document and the frames within it as a parent-child relationship. The window that the framesetting document loads into is the parent window, and each frame defined within that parent window is a child frame.

In addition to this parent-child analogy, let's introduce the concept of the top object. While a child frame's parent is the object immediately above it in the object hierarchy, the top object is the object that is highest in the hierarchy that the child frame belongs to. Figure 2 illustrates the concept of parent and top objects in a multigenerational model of frames and framesets.

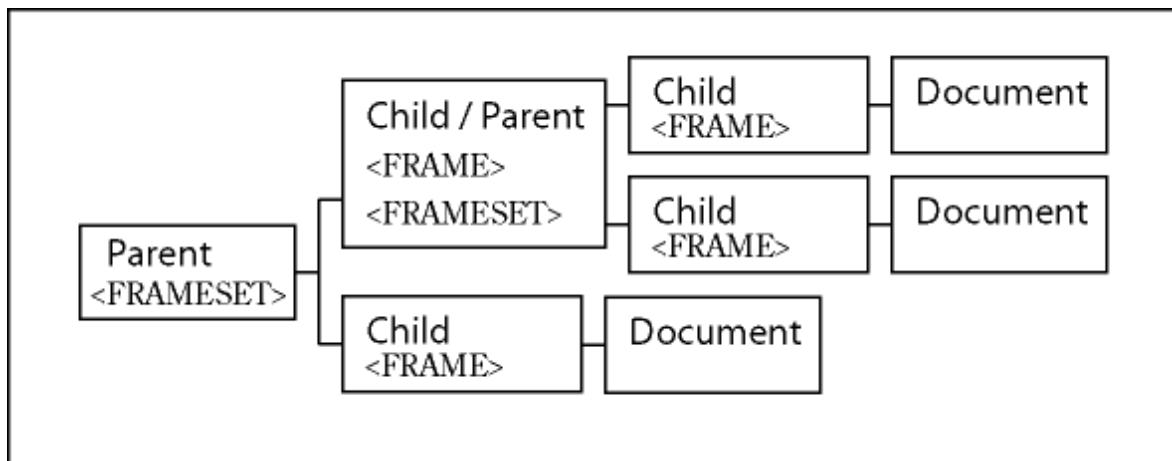


Figure 2. This multigenerational object model represents three generations of window objects.

Understanding the relationships among frames is essential if you hope to have them communicate. As with windows, you can reference variables, functions, and objects in any relative frame.

**TIP:** The key to referencing another frame is to begin the reference with the window object that the two relatives have in common and work your way to your destination object.

*See*

***FrameReference\_Index.html***

The following example illustrates how to reference a child frame from within another relative frame. The object model resembles the one shown in Figure 2. The top frameset defines two frames in columns; each references existing HTML documents that are in the same local directory.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
  "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
  <head><title>Top Frameset</title></head>
  <frameset cols="5%, 95%">
    <frame name="indexFrame"
          src="FrameReference_Index.html">
    <frame name="contentFrame"
          src="FrameReference_Content.html">
  </frameset>
</html>
```

*See*

***FrameReference\_Index.html***

The document in the left frame defines a function that will list the numbers one through five vertically in the frame, as links. When clicked, the links each write a string to another frame that is lower in the document hierarchy. This frame is defined in the code listing called FrameReference\_Content.html. The string that defines the content for the left frame is built incrementally as you iterate through a loop, and the content is displayed within the body of the page. The code for the left frame is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
        "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<html>
<head><title>Index</title>
<script language="JavaScript" type="text/javascript">
    function createIndex() {
        var html = "";
        for (i = 1; i <= 5; i++) {
            html += "<a href=\"javascript: ";
            html += "parent.contentFrame.botFrame.";
            html += "document.write(\"";
            html += "'You selected " + i + '");
            html += "parent.contentFrame.";
            html += "botFrame.document.close() ";
            html += "\">" + i + "</a>";
            html += "<br>";
        }
        return html;
    }
</script>

</head>
<body>
<script language="JavaScript">
    document.write(createIndex());
</script>
</body></html>
```

*See*

**FrameReference\_Content.html**

Finally, the top frameset's right frame loads another frameset. This frameset defines two frames that are displayed in rows. Both of the new frames are defined dynamically within the <frame> tag. The top frame simply prints static text, while the bottom frame is initially blank. This bottom frame will be acted upon by the top frameset's left frame and will display a message based on the user's selection.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
  "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
  <html>
    <head><title></title>
    </head>
    <frameset rows="30%,70%">
      <frame name="topFrame"
            src="javascript:'<html><body><h1>' +
                  'Top Right Frame</h1></body></html>'">
      <frame name="botFrame"
            src="javascript:'<html></html>'>
    </frameset>
  </html>
```

With all three files from the previous example in place, you can load the FrameReference\_Top.html file to view all three frames. Clicking on the links in the left frame writes a string to the bottom left frame. You may have found it difficult to read the statement, as it was broken into several segments in order to build the string. The key reference, from the left window of the top frameset to the bottom frame in the frameset within the top document's right frame is:

```
parent.contentFrame.botFrame.document.write(
  "'You selected " + i + "'");
```

So, the most common ancestor of both frames is indexFrame's parent. From there you can work down the object hierarchy until you get to botFrame, where you can then invoke any methods of that frame.

## iframes

While frames can be useful in improving the navigation options of a page, they are inevitably constrained to hugging the edge of a document, which limits your design options. An alternative solution is to use inline frames, or iframes for short.

Iframes behave much like a frame does, enabling you to display content from another document or pass information between the parent document and the iframe itself, but they have many differences as well. First, iframes can be positioned within the `<body>` tags of a document, just like an image. Secondly, iframes are defined entirely by their own `<iframe>` tag, removing the need for a `<frameset>` or `<frame>` tag.

A typical definition for an iframe specifies its dimensions (measured either in pixels or in a percentage of the bounding document) and the source for the content to display. As with a regular frame, it is also a good idea to give the iframe a name, so that you can reference it elsewhere in your scripts. Here is an example:

```
//This defines an iframe's dimensions in pixels.  
<iframe width=200 height=200 src=http://www.appdev.com>  
    name="newIframe">  
  
</iframe>
```

Or:

```
//This defines an iframe's dimensions in a percentage of  
//the bounding container.  
<iframe width=40% height=40% src=http://www.appdev.com  
    name="newIframe">  
</iframe>
```

**NOTE** The `<iframe>` tag requires a closing `</iframe>` tag.

# Summary

- Spawn a new window from within a document by using the `window.open()` method.
- Reference and control your new window by assigning the return value of the `window.open()` method to a variable.
- An originating window can pass information to a window that it spawns and vice versa.
- Create modal dialog boxes in Internet Explorer 5+ that restrict how a user interacts with your pages.
- Frames allow you to open multiple documents within one window.
- Frames are window objects, and the standard properties and methods of a window object apply to them.
- Framesets manage the relationships among frames within a document.
- Frames can themselves load a new frameset.
- Frames can interact with other frames that derive from a common ancestor window by following the hierarchy of window references.
- iframes are a stylistically more flexible alternative to frames.
- iframes can be defined inline, like an image.

# Questions

1. How do you open a new window from within your scripts?
2. True/False: You must always assign the return value of `window.open()` to a variable.
3. What is the easiest way to dynamically create content for a spawned window?
4. Which property enables a new window to reference the window that created it?
5. What is the term for a frame that is created by a frameset?
6. Which property enables a frame to access the document that created it?
7. Which property enables a frame to access the highest object in its object hierarchy?
8. True/False: An iframe is constrained to the edges of the document area of a window.

# Answers

1. How do you open a new window from within your scripts?  
**By invoking the window.open() method.**
2. True/False: You must always assign the return value of window.open() to a variable.  
**False. It is perfectly acceptable to not assign the return value to a variable if you don't intend to reference the new window in your scripts.**
3. What is the easiest way to dynamically create content for a spawned window?  
**By building a dynamic HTML string that defines the content of the new page and writing it to the new window through its window.document.write() method.**
4. Which property enables a new window to reference the window that created it?  
**The opener property**
5. What is the term for a frame that is created by a frameset?  
**Child frame**
6. Which property enables a frame to access the document that created it?  
**The parent property**
7. Which property enables a frame to access the highest object in its object hierarchy?  
**The top property**
8. True/False: An iframe is constrained to the edges of the document area of a window.  
**False. An iframe can float anywhere within the document that you chose to define it.**

# Lab 7: Windows and Frames

**TIP:** Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You will find all the code in **Windows and Frames.html**, **monthFrame.html**, **SimpleCalendar.html**, **SimpleCalendar.js**, and **SimpleCalendar.css** in the same directory as the sample project. To avoid typing the code, you can cut/paste it from the source files instead.

# Lab 7 Overview

In this lab you will learn how to use frames and spawned windows effectively. Through the course of the lab, you will improve upon a simple calendar by adding a date selection frame and pop-up info about a limited list of holidays.

To complete this lab, you will need to work through two exercises:

- The Date Selection Frame
- Important Date Pop-Up Info

Each exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

# The Date Selection Frame

## Objective

In this exercise, you'll create a frameset to contain your date selection frame, and a frame to contain the calendar. You'll also create the controls for your date selection frame, and the functions that will update the calendar frame.

## Things to Consider

- Rather than overwriting an entire html page, you can overwrite just the html within a <div> tag by modifying the innerHTML property of the <div> tag. The way in which Internet Explorer and Netscape Navigator support this property differ, so examine **monthFrame.html** to get a feel for the differences. This can prevent you from overwriting functions and variables that you need to persist within the page.
- The scope of items in a .js file is the same as if the text in that file were substituted for the script tag in which it is referenced. Therefore, accessing items in a .js file is the same as accessing those in the containing XHTML file.

## Step-by-Step Instructions

1. Open the incomplete **AdvancedCalendar.html** file and look for the comment that indicates Exercise 1.
2. In the space noted by the comment, add the XHTML code necessary to create a frameset that defines two frames, organized in columns. The left frame should use the provided **monthFrame.html** file as its source, while the right frame should use the provided **SimpleCalendar.html** file as its source. The exact dimensions of each frame are up to you, but you should make sure that each frame fully displays its controls.

```
</head>

<frameset cols="15%,85%">
    <frame name="monthFrame"
        src="monthFrame.html">
    <frame name="calendarFrame"
        src="SimpleCalendar.html">
</frameset>

</html>
```

3. Next, open the incomplete **monthFrame.html** file and locate the comments that indicate Exercise 1. Within the function **populateDays()**, you must first determine how many days are in the selected month so that you can populate the select list with the available days of the month. There is already a function called **getMonthLen()** in the provided **SimpleCalendar.js** file that does this, so rather than rewriting it, simply call the existing function and assign its return value to a new variable named **monthLength**.

```
var monthLength =
    parent.calendarFrame.getMonthLen(
        document.controls.month.selectedIndex,
        document.controls.year.value);
```

4. Examine the function **replaceCalendar()**, which is intended to replace the HTML within the **<div>** tag of **SimpleCalendar.html** with code that defines the new calendar. This code is different than the existing statement within the original file, in that it passes the date specified by the controls in the left frame into the call to **buildCalendar()** in the right frame. The **replaceCalendar()** function has already defined the left side of the assignment statement needed, which will be different based on which browser is being used to view the page. You must combine the string in the **divRef** variable with a string that defines the right side of the assignment statement. This will be the actual text that you want to go in the **<div>** tag of the right frame, which extends the original, by adding a date value.

```
eval(divRef + " = top.calendarFrame.buildCalendar(" +
    "new Date(" +document.controls.year.value +
    ", " +
    document.controls.month.selectedIndex +
    ", " + document.controls.day.value + ")");
```

5. Open the **AdvancedCalendar.html** file in your browser and test the functionality of both frames (see Figure 3).

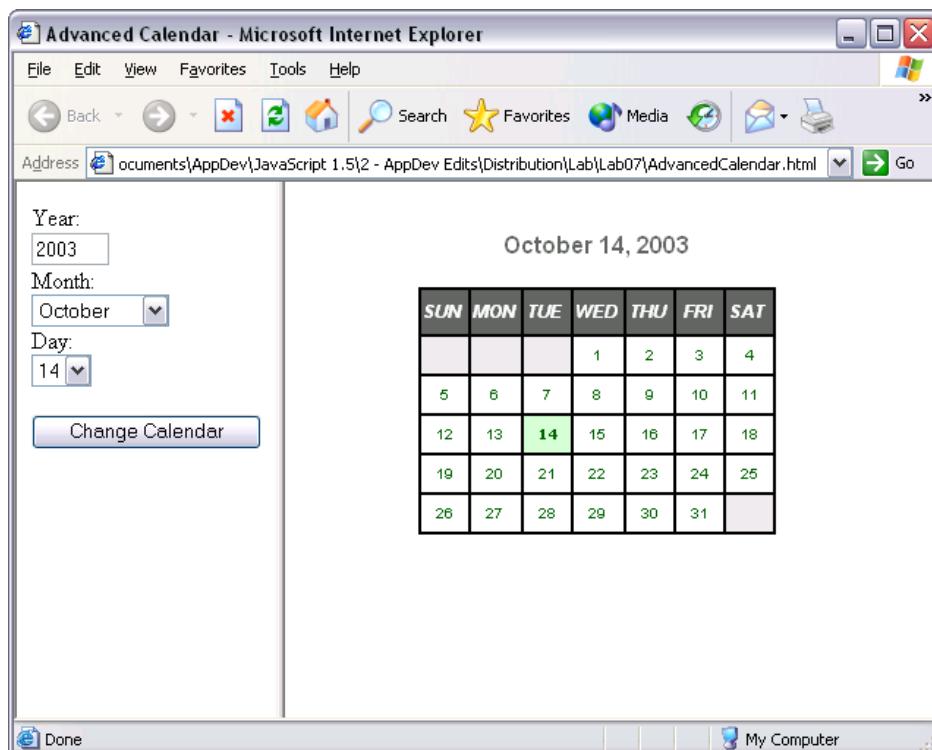


Figure 3. The Advanced Calendar in Internet Explorer.

# Important Date Pop-Up Info

## Objective

In this exercise, you'll create an array of information about important dates and display the info for each date in a new window whenever the user clicks on that date.

## Things to Consider

- This example only considers events that happen on the same date every year. If you want to consider roving holidays, feel free to adapt the example once you are done.

## Step-by-Step Instructions

1. Open the incomplete **SimpleCalendar.js** file, and look for the comments indicating Exercise 2.
2. Examine the **dateInfo** array at the beginning of the file, which is the data structure that the important date information is to be stored in. The file provides you with three sample entries, but you should feel free to add your own dates, such as important birthdays. The format that we set for each record in the array follows the pattern:  
**date(mmdd):Name(text):Description(text)**. Examine the entries provided for a practical example. Since all of the information to be stored in the array will be strings, this will make it easier for you to manipulate each record and access multiple pieces of information.
3. Complete the **displayDateInfo()** function defined at the end of the file. The first thing that you will want to do is to open a new window, which won't need much space to display the date information:

```
//Open the new window.  
var infoWindow = window.open("", "infoWindow",  
    "height=200, width=400, dependent");
```

4. Notice that **displayDateInfo()** takes three parameters, **m**, **d**, and **y**, which represent the month, day, and year of the date that the user selects. Since the first portion of each record in the **dateInfo** array is the month and the

day in mmdd format, you may need to pad one or both of the **m** and **d** parameters to be two digits each.

**TIP:** Since the parameter values are being passed as integers, you may want to force a conversion to string values, or else any attempt to concatenate the two strings with the addition operator will add the integer values of the two parameters.

```
//If m and d have only a single digit, pad them with a
//leading 0.

var mStr = "" + m;
var dStr = "" + d;
if (mStr.length < 2) {
    mStr = "0" + mStr;
}
if (dStr.length < 2) {
    dStr = "0" + dStr;
}
```

5. Next, within **displayDateInfo()**, begin constructing a string that contains the XHTML for the new page. Focus on the tags up to the **<body>** tag, ensuring that you specify a title:

```
//Construct the HTML string
var infoHTML = "";
infoHTML += "<?xml version=\"1.0\" encoding=\"";
infoHTML += "\"utf-8\"?>";
infoHTML += "<!DOCTYPE html public";
infoHTML += "\"-//W3C//DTD XHTML 1.0 Strict//EN\"";
infoHTML += "\nhttp://www.w3.org/TR/xhtml1/";
infoHTML += "DTD/xhtml1-strict.dtd\">";
infoHTML += "<html xmlns=\nhttp://www.w3.org";
infoHTML += "/1999/xhtml\" ";
infoHTML += "xml:lang=\"en\" lang=\"en\"><head>";
infoHTML += "<title>Date Details</title>"
infoHTML += "<body>";
```

**Lab 7:**  
**Windows and Frames**

---

6. Next, add a string to the XHTML string that displays the date in an <h3> tag.

```
//Display the date, no matter what.  
infoHTML += "<h3>" + m + "/" + d +  
"/" + y + "</h3>;
```

7. The next step within **displayDateInfo()** is to write a for loop that iterates through the **dateInfo** array and check if the mmdd portion of the string in any of the records match the month and day that were passed in as parameters. If there is a match, then you should add the date info to the XHTML string and break out of the function. You can get the special name of the date, as well as the descriptive text for the date, out of the record in discrete elements by using the **string.split()** method on any matching record in dateInfo.

```
//Iterate through the dateInfo array and check for a  
//match on the date.  
for (var i = 0; i < dateInfo.length; i++) {  
    if (dateInfo[i].substring(0, 4) == mStr + dStr) {  
        //If a matching record is found, split the  
        //strings within based on the ":" delimiter  
        //and store the values in a new array.  
        var dateRecord = dateInfo[i].split(":");  
        //Add the date details to the HTML string.  
        infoHTML += dateRecord[1];  
        infoHTML += "<br>";  
        infoHTML += dateRecord[2];  
        infoHTML += "<br>";  
  
        break;  
    }  
}
```

8. Next, finalize the XHTML string to close the <body> and <html> tags:

```
infoHTML += "</body>";  
infoHTML += "</html>";
```

9. To finalize the body of **displayDateInfo()**, add a statement that writes the XHTML string that you constructed to the new window. Don't forget to close the document stream.

```
//Write the information to the new window.  
infoWindow.document.write(infoHTML);  
infoWindow.document.close();
```

10. Finally, you need to have the calendar call your new function whenever the user clicks on a date. This is done in two places, as the cells of the first row are generated separately from the rest of the rows. Look for the sections where the HTML string that defines the cells is generated, and change the **onClick** event for both to call **displayDateInfo()**, passing the required parameters. Be careful with embedded quotes in your string construction.

```
html += "\\" onmouseover="" +  
    "this.style.cursor='hand'" +  
    " \" onclick=\"displayDateInfo(" +  
    year + ", " + (date.getMonth() + 1) +  
    ", " + (c + 1) + ")" " +  
    "\">>" + (c + 1) + "</td>";
```

// And:

```
html += "\\" onmouseover="" +  
    "this.style.cursor='hand'" +  
    " \" onclick=\"displayDateInfo(" +  
    year + ", " + (date.getMonth() + 1) +  
    ", " + p + ")" " +  
    "\">>" + p + "</td>";
```

11. Test the exercise by loading **AdvanceCalendar.html**, setting the date to one of the special dates that you added to the array, and clicking on that date in the calendar. If you were successful, you should see a description for that day pop-up in a new window (see Figure 4). If there are no details for that date in the array, then just the date will display in the new window.

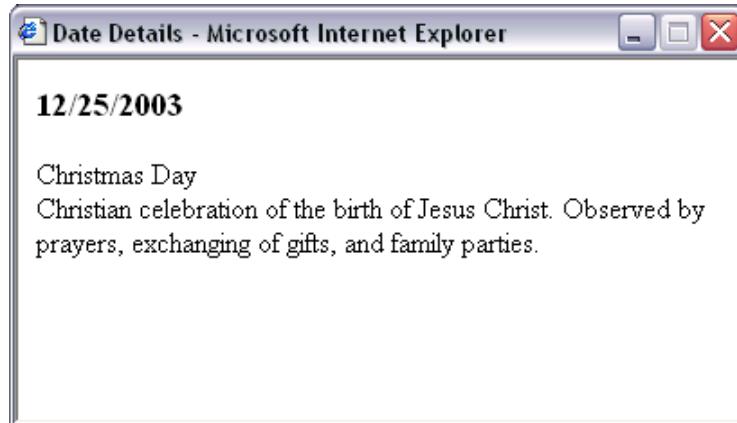


Figure 4. Pop-up display for Christmas.

# Event Handling

## Objectives

- Understand the key differences in the most popular browser event models.
- Discover challenges and solutions of scripting cross platform events.
- Learn to use simple form element event handlers, and expand on that knowledge.
- Script event handlers using the Event() object.

# Popular Browser Event Models

When you create interactive Web pages in JavaScript, it is important to understand the concept of the browser event model. Most every graphical user interface (GUI) requires that you understand how its event model works to program or script for it properly.

Back in the good old days of command line interfaces, the only input that programs had to respond to occurred when the user pressed a key on the keyboard. Today, all of the major operating system platforms and most software programs have some sort of graphical user interface that relies on the power of events. It is essential to understand how the underlying event models work in order to deal with mouse clicks, timed events, and machine level events, as well as events that are generated by the keyboard, USB devices, and other input mechanisms.

JavaScript is both good and bad for dealing with the concept of event handling. It is good because, with the advent of the version 4 browsers, you can script, capture, and manipulate event objects. It is bad because you have to contend with three different event models to ensure that your scripts work with each of the major browser platforms.

Figure 1 shows an alert box that was triggered by a script associated with a button's onClick event in the Mozilla browser.

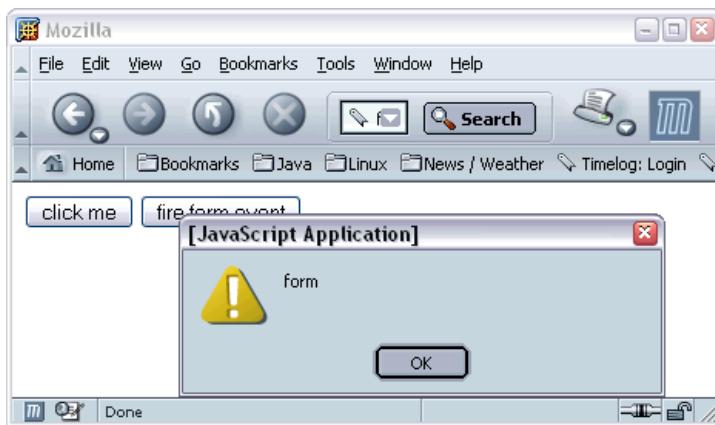


Figure 1. A Mozilla event fired.

## The Sequence of Events

Events typically originate in the operating system, which is responsible for handling all the interactions between the user and the computer. When an event occurs, such as a mouse click or a key press, the operating system recognizes the event and notifies the browser of the event. Once the browser is

aware of the event, it notifies any HTML elements on the page that have an event handler of the appropriate type to deal with that event.

The event handler is an attribute of an HTML element and can specify a JavaScript function to call when the event arises. You activate these event handlers by listing them as attributes in the tag for the HTML element that you want to associate them with. For instance, a button object has many available event handlers that you can harness, such as those for the onClick, onFocus, and onBlur events.

An operating system usually buffers events and waits until one is handled before releasing another. As a result, the browser only receives one event at a time, which simplifies the task of handling them in your code.

## **The Event Object**

One thing that these browser platforms have in common is that they create an event object whenever they receive an event notification from the operating system. Events carry useful information, which references the event that occurred. For example, if a user clicks the mouse button, the event contains information about which button was clicked and the exact coordinates on the screen where the click occurred.

An event object's lifespan can be very short. If the page has no event handlers for the specific type of event, the event object's life begins and ends almost instantaneously. However, if an appropriate event handler exists, and the event object is accessed via JavaScript, it can be handled through any number of functions and kept alive until it reaches the conclusion that you scripted for it.

## **Bubbling vs. Capturing Events**

In the three major browser platforms, there are two ways in which events travel through the document object model (DOM): bubbling and capturing. The difference between bubbling and capturing has to do with the level at which the event is first handled. For the major browser platforms, the window object is at the outermost level of the DOM hierarchy, followed by the document object, the form object, and finally the HTML element objects.

A capturing event model starts at the top of the DOM hierarchy and makes the event available to the window object first. From there, the event passes down through the hierarchy to the document, the form, and the HTML element object levels. In this model, you can actually capture an event at a higher level before it reaches the intended target, such as an HTML button, and handle it without ever passing the event down to the target object.

In a bubbling event model, things flow in the opposite direction. This means that the event can be caught first by the lowest level objects on the page, which

are typically HTML elements, such as a button or a text field. From there, the event is passed up to the next higher level such as the form object, and up through the levels of the DOM hierarchy until it reaches the window object.

## The Window and Document Objects

The window and document objects are not defined within the HTML of a Web page. However, there will be times when you want to handle events at the document or window level. JavaScript provides implicit access to these objects through the following keywords: window and document. For example, if you want to execute a function when the user clicks on the window or document, you would assign the function to that object's event handler, as shown in the following code snippet:

```
<script language="JavaScript">
    window.onclick = winFunction;      //fires on window click
    document.onclick = docFunction;   //fires on document click

    function winFunction() {
        alert("Window was clicked");
    }

    function docFunction() {
        alert("Document was clicked");
    }
</script>
```

## Bubbling Events in Internet Explorer 4+

In Internet Explorer 4+, the browsers employ the bubbling event model. When an event occurs, the event object is first passed to the HTML element that captured it, and is subsequently passed up through each of the objects above it in the DOM hierarchy. For example, when a button on a form is clicked in Internet Explorer 4, that button's event handler captures the event and then passes it to the form that contains the button. From there it is passed to the body object, then to the document object, and finally all the way up to the window object.

The bubbling action in Internet Explorer 4 is very similar to that of the object model defined by the World Wide Web Consortium (W3C), which is also implemented in Netscape Navigator 6 or Mozilla 1.0. However, this is the reverse of the object model defined in Netscape Navigator 4, which requires explicit event capturing, as you will see later in the chapter.

See **bubbling example.html**

One way to test event bubbling is to watch an event travel through the hierarchy within the window. Since the window and document objects are not defined within the HTML page, you need to script them:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
<head>
<script language="JavaScript">
    window.onclick = wine;
    document.onclick = doc;

    function wine() {
        alert("window");
    }
    function doc() {
        alert("document");
    }
</script>
</head>
<body>
<form name="form1" onclick="alert('form')">
    <input type="button" value="click me" name="button"
        onclick="alert('button');"/>
</form>
</body>
</html>
```

In this example, the event is initially handled by the button's onclick event handler, as shown in Figure 2, and then bubbles up through the higher-level DOM elements. It is a good idea to test this in various browsers to see how the effect works. While testing, try clicking on the button and around the button. You might want to put in other form elements and event handlers to see how they react.

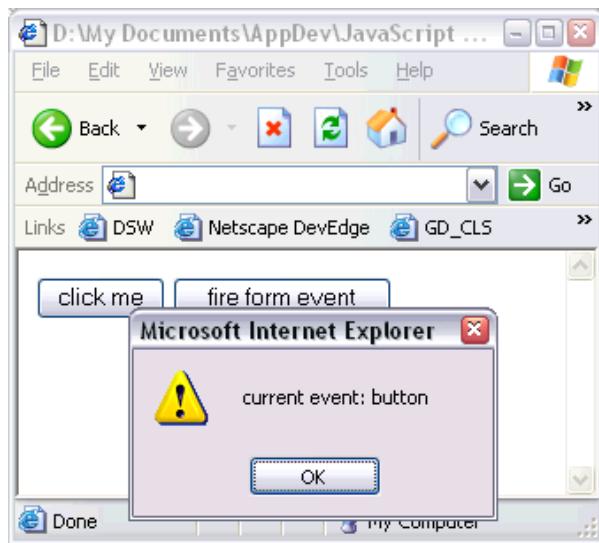


Figure 2. Internet Explorer 6 event description.

In some circumstances, you may want your script to stop the bubbling process at a certain level. Internet Explorer's event object has a property, `event.cancelBubble`, which is just the ticket to stop the event bubbling action. At the point in the hierarchy where you want to stop bubbling, simply set the `event.cancelBubble` property to true. In the following example, the event bubbling process will stop after the button's `onClick` event fires and the alert runs:

```
<form name="form1" onclick="alert('form')">
  <input type="button" value="click me" name="button"
    onclick="alert('button'); event.cancelBubble =
      true"/>
</form>
```

Although this property was defined in Internet Explorer 5.5, it also works fine in tests using Mozilla 1.3 and Opera 7.11. If your users are working with the more current browser versions, this is a good way to contain the event and prevent higher-level objects from firing handlers for it.

You can also have the event handlers for one object fire events for another object on the page. You do this by calling that object's `fireEvent()` method and passing it the name of the specific event handler that you want to trigger, along with a reference to the event:

```
<input type="button" value="fire frm event" name="button1"
  onclick="document.form1.fireEvent('onclick', event)"/>
```

In the previous example, when the “fire frm event” button is pressed, it causes the form’s onClick event handler to fire. On a compatibility note, this feature works fine in Mozilla 1.3. In Opera 7.11, the form’s onClick event fires, but it also generates the following error message:

```
Event thread: onclick
Error:
name: TypeError
message: Statement on line 1: Expression did not evaluate
to a function object: document.form1.fireEvent
Backtrace:
  In unknown script
    document.form1.fireEvent("onclick", event);
  At unknown location
    {event handler trampoline}
```

## **Netscape Navigator 4 Event Capture Model**

The number of users who are still working with Netscape Navigator 4 has greatly diminished. Depending on the desired reach of your Web site, however, cross-browser compatibility may still be an issue, so it is important to try to maintain compatibility as much as possible. Unlike the Internet Explorer 5+ browsers, Netscape Navigator 6+ has retained many of the event capturing methods and properties used in Netscape Navigator 4, for backward compatibility. This is just one more reason why it is important to understand how the different event models work.

Netscape Navigator 4 uses the capture event model, which means that events are first handled at the top level of the DOM hierarchy, the window, and then are passed down to the lowest level, the HTML elements.

One of the handy features of the Netscape Navigator 4 event object is that it carries a reference to its intended target element, which makes capturing events for specific form elements much easier. If you were to click on a button and capture the event at the document level, you could execute some setup code and then explicitly pass the event to its original target.

Another difference in the Netscape Navigator 4 event model is that you must first enable the event-capturing feature before you can use it, as it is not enabled by default. A further limitation is that the window, document, layer, and HTML elements are able to capture events, while the form object cannot.

See [Netscape4 Example.html](#)

The window, document, and layer elements all have a method, `captureEvents()`, that enables the event capture capability for each one. In this event model, captures are more explicit than the Internet Explorer 5.5+ or Netscape Navigator 6+ event models. When you call the `captureEvents()` method, you supply parameters to it that specify which events to capture. Any type of event that is not specified will continue to be passed down through the DOM hierarchy:

```
<script language="JavaScript">

    window.captureEvents(Event.CLICK | Event.KEYPRESS);

    window.onClick = windowClick;
    window.onKeyPress = windowKeyPress;

    function windowClick() {
        alert("clicked!");
    }

    function windowKeyPress() {
        alert("key pressed!");
    }

</script>
```

In the preceding code example, notice how you can set the code to capture multiple events by passing the corresponding event constants to the `captureEvents()` method separated by the pipe (|) symbol. Figure 3 illustrates the resulting behavior when the `window.onClick` event is triggered in Netscape Navigator 4.

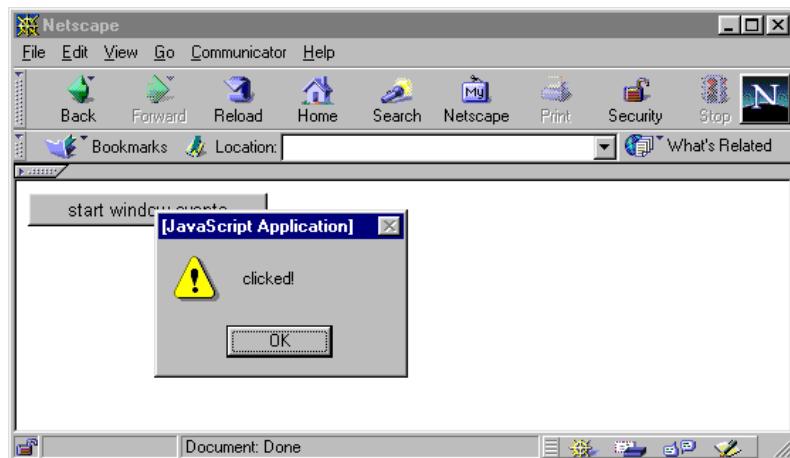


Figure 3. Netscape Navigator 4 event example.

When you script events in Netscape Navigator 4, you may want to catch an event that was triggered with a modifier key held down, such as **CTRL**, **ALT**, or **SHIFT**. In this case, some alternate form of behavior might be required, which requires your script to notify the target object so that it can respond appropriately. One way to accomplish this is with the `routeEvent()` method, which allows you to pass an event to the intended target after your modifications have been made:

```
function testClick(cevent) {  
    if (cevent.modifiers & Event.ALT_MASK) {  
        alert("alt-clicked!");  
    } else if (cevent.modifiers & Event.SHIFT_MASK) {  
        alert("shift-clicked!");  
    } else if (cevent.modifiers & Event.CONTROL_MASK) {  
        alert("ctrl-clicked!");  
    } else {  
        alert("regular click");  
    }  
}  
  
function windowClick(e) {  
    if (e.target.name == "button1") {  
        routeEvent(e);  
    } else {  
        alert("caught a click");  
    }  
}
```

**NOTE** In Netscape Navigator 4, when you assign a function to an event handler, only specify the function name; do not include parentheses or parameters. However, you should be aware that the event object is still passed to the function. So in the `windowClick` function, `(e)` is required to capture the event that is implicitly being passed to the function.

Figure 4 uses the `Event.ALT_MASK` constant to indicate when the button was clicked while the user held down the **ALT** key.



Figure 4. Netscape Navigator 4 using routeEvent().

Finally, you may want an object other than the target object to handle the event. You do this via the handleEvent() method, which works a little differently than the routeEvent() method. When an event occurs, you can choose to hand that event off to another object for processing, via that object's handleEvent() method. If the object that receives the event has a handler for the event type, it will accept the call. For example, your window object is set to catch the onClick event. When it receives the event, it lets a text object handle the event:

```
function windowClick(e) {  
    if (e.target.name == "button1") {  
        routeEvent(e);  
    } else if (e.target.name == "checkbox") {  
        alert("Caught a click over checkbox, handing over  
        event to text.");  
        document.form1.text.handleEvent(e);  
    } else {  
        alert("caught a click");  
    }  
}
```

In the preceding code snippet, the text object is an element on the form that receives the click event, which fires text's onClick event handler, if one is defined.

## Events in Netscape Navigator 6+ (W3C Compliant Version)

With the release of Navigator 6, the Netscape designers wanted the DOM to maintain compatibility with older systems, but also allow new functionality to be added. As a result, the Netscape Navigator 6+ model implemented both the bubbling and capture event models. The bubbling method is used by default, but you can enable the capture method.

Netscape Navigator 6 implements the capture method by providing an `addEventListener()` method. The idea is to call this method from the object that you want to designate as the event handler, and instruct your script to direct it to a function when it receives the event, whether by capture or by bubbling. In fact, you can even have it launch different functions depending on which method of event propagation is used.

```
document.form1.addEventListener("[event type]",
    [function to call], [Boolean, true for capture, false
    for bubble]);
```

```
document.form1.addEventListener("click",
    fireFunction, false);
```

If you need to remove the event listener at some point in the page's lifetime, you can call the `removeEventListener()` method and pass it the same parameters that were passed to the `addEventListener()` method.

Similarly, the Netscape Navigator 6 model allows you to stop event bubbling, also known as propagation, by providing a `stopPropagation()` method. This has the same effect as Internet Explorer's `event.cancelBubble()` method. And finally, Netscape Navigator 6 contains a `dispatchEvent()` method—which is similar to Internet Explorer 4's `fireEvent()`—that allows you to control events on other objects.

Navigator 6's `dispatchEvent()` method by default passes the event object as its parameter to the target object. This is much more convenient than having to refer back to the parent or passing a bunch of parameters down.

See [Netscape6Plus Example.html](#)

The following example shows how the event is caught during the capture phase of its lifecycle and then redirected to the hidden object in `form2`. Subsequently, it illustrates how the event is captured by the button, then by the form via the bubbling process, and finally how the event is passed to a different hidden object:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="en" lang="en">
<head>
<script language="JavaScript">
function catchFormBubble(evt) {
    alert("Form caught 'bubble' event, sending to " +
        "'hidden'.");
    document.form2.hidden.dispatchEvent(evt);
}

function catchFormCapture(evt) {
    alert("Form caught 'capture' event, sending to " +
        "'hidden1'.");
    document.form2.hidden1.dispatchEvent(evt);
}

function setListener() {
    // 'false' parameter means this will catch bubbling
    // events only. Otherwise, it will catch capture
    events.
    document.form1.addEventListener("click",
        catchFormBubble, false);
    document.form1.addEventListener("click",
        catchFormCapture, true);
}
</script>
</head>
<body onload="setListener()">
<form name="form1">
    <input type="button" value="Start Event" name="btn"
        onclick="alert('caught by button...')"/>
    <input type="button" value="Prevent Capture"
        name="btn"
```

```

        onclick="document.form1.removeEventListener" +
        "('click', catchFormCapture, true)"/>
<input type="button" value="Prevent Bubble" name="btn"
       onclick="document.form1.removeEventListener" +
       "('click', catchFormBubble, false)"/>
</form>

<form name="form2">
<input type="hidden" name="hidden"
       value="event reached hidden object in form2!">
<input type="hidden" name="hidden1"
       value="event reached hidden1 object in form2!">
</form>
</body>
</html>

```

**NOTE** In this example, the eventListeners are added *after* the page is loaded. This ensures that the event is assigned after the object is created. In addition, the “Prevent Capture” and “Prevent Bubble” buttons are included to demonstrate how a specific event listener can be removed dynamically in your script.

The previous example was tested in Mozilla 1.3, Opera 7, and Internet Explorer 6. Internet Explorer 6 fired only the button’s onClick method, since it does not support W3C methods for event handling. However, Opera 7, which tries its best to be W3C compliant, ran the example without a hitch.

A good way to visualize bubbling and capturing events is to envision an HTML page as a set of layers. Consider the window as the top layer, proceeding down to the document layer and the form layer, to the bottom layer, which consists of HTML elements.

Now think of the event as a super-bounce rubber ball. On the way into the layers, it is a capture event; once it hits bottom and bounces back out through the layers, it is a bubble event. Therefore, in the previous example, the ball hits the form eventListener and fires the form’s capture eventListener definition. It proceeds down to the button object and fires its onClick event. It then bounces back up where it is caught by the form’s bubble event listener, which is fired, and then the ball flies away into oblivion.

Figure 5 illustrates the “rubber ball” effect of the bubbling and capturing process.

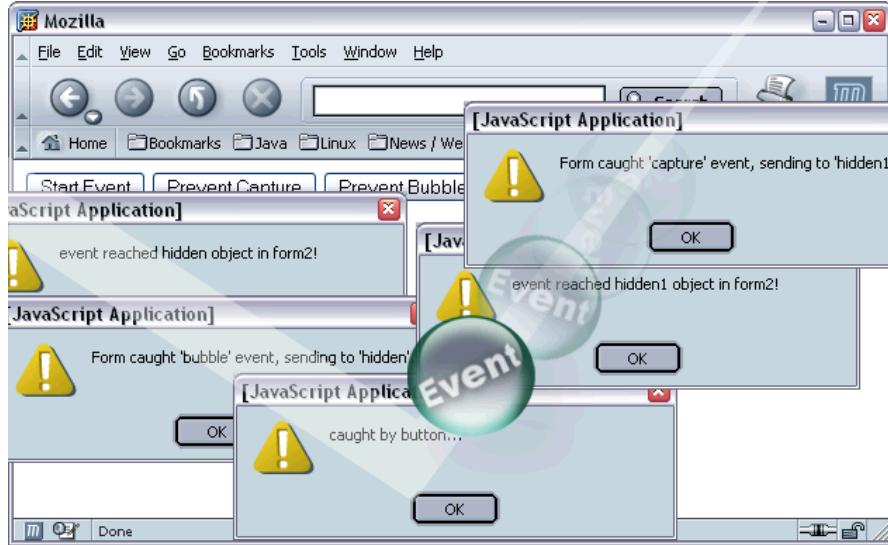


Figure 5. The capture and bubble event model illustration.

It is usually helpful to try to visualize what is happening in any event model so that you do not lose track of which object is handling which event. For example, suppose you create a bubble event listener that listens for onClick events on a form. Now, suppose that form passes the event back to the button. In this case you created a looping effect, because every time the event is fired again on the button, it is passed to the form event listener, and so on.

Now that you have seen the three major event models, you might ask, “What is different about the actual event object?”

# Event Objects

When you script event-handling routines, it will come in handy to know what the various properties and methods of the event object are. In the previous section, you learned that three different event models are used in today's popular browsers. Fortunately, when it comes to accessing the actual event objects, there are only two types: the Internet Explorer 4+ event object and the Netscape Navigator 4+, or W3C, event object.

## The Static Event Object

The static event object is the model for events in the W3C event model, meaning it is only relevant to Netscape Navigator 4+ and Opera. The static object in Netscape Navigator 4 is useful for comparing properties, such as modifiers. Modifiers include the **ALT**, **CTRL**, and **SHIFT** keys, and you can use the static Event objects to determine whether one of those modifiers was pressed when the event occurred.

## Standard Event Objects

Standard event objects are created from the static object. When scripting for the standard event objects you must be aware of two things: first, event objects are not always passed automatically, as they are in Netscape Navigator 4+; second, all of the properties and methods of standard event objects are accessed differently. This means that it is easy to receive an event object in script, since you only need to know two different ways to capture an event, but it is more complex to actually script their various properties and methods once you've captured them.

In Internet Explorer you simply use the `window.event` property to access the event object. (You do not have to access the event object explicitly in Netscape Navigator 4, because the browser passes the event object to the event handler implicitly.)

Accessing the Internet Explorer event object is simple:

```
// Internet Explorer 4+
window.event
```

## Referencing the Netscape Navigator 4+ Event Object

*See NN4Event Object.html*

Unfortunately, accessing event objects takes a little more expertise in the Netscape Navigator 4+ model. In Internet Explorer 4+, you can access the event anywhere in the script using window.event, which is globally the current event. However, in Netscape Navigator 4+ the event object is only available in the context of an HTML element's event handler. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
        "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      lang="en">
<head>
<script language="JavaScript">
    function sendEvent(evnt) {
        alert("Target: " + evnt.target.name);
    }
</script>
</head>
<body>
<form name="form1">
    <input type="button" value="Launch Event" name="btn"
          onclick="sendEvent(event)"/>
</form>
</body>
</html>
```

Executing the code in the preceding example results in the output shown in Figure 6.

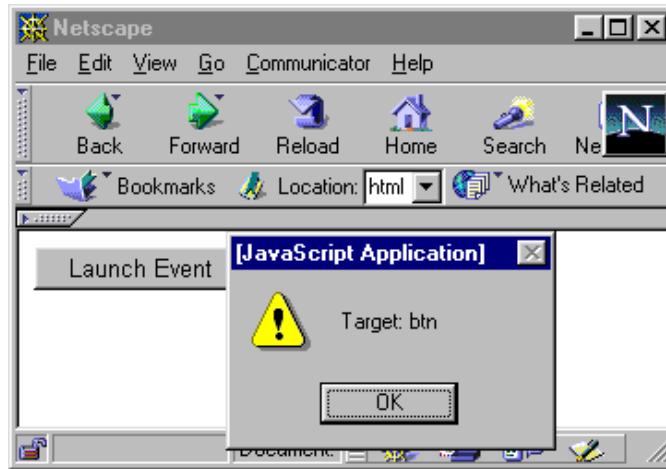


Figure 6. A Netscape Navigator 4+ event passing.

In contrast, the following code will generate an error because the event is not defined:

```
<head>
<script language="JavaScript">
function showEvent() {
    alert("Target: " + event.target.name);
}
</script>
</head>
<body>
<form name="form1">
    <input type="button" value="Show Event" name="btn1"
        onclick="showEvent ()"/>
</form>
</body>
```

See  
[eventobject.html](#)

The other way to let a function access an event in Netscape Navigator is to pass it via assignment. For example, when you define an eventListener to an object (Netscape Navigator 6+) or you directly define the event for the object (Netscape Navigator 4), the current event is passed automatically; you just have to catch it. Consider the following code:

```
<head>
<script language="JavaScript">
function showEventNN4Cap(evnt) {
    alert("NN4 Target: " + evnt.target.name);
}

function showEventNN6Cap(evnt) {
    alert("NN6 Target: " + evnt.target.name);
    evnt.stopPropagation();
}

function setupListeners() {
    document.captureEvents(Event.CLICK);
    document.onclick=showEventNN4Cap;
    if (parseInt(navigator.appVersion.charAt(0)) > 4) {
        document.onclick=null;
        document.addEventListener("click",
            showEventNN6Cap, true);
    }
}

</script>
</head>
<body onload="setupListeners()">
<form name="form1">
    <input type="button" value="Send Event" name="btn"
        onclick="alert('button');"/>
    <input type="button" value="Show Event" name="btn1"
        onclick="alert('button1')"/>
</form>
</body>
```

The preceding example illustrates how you can provide access to the event object in your script, regardless of whether it is running in the Netscape Navigator 4 or Netscape Navigator 6+ browsers. The setupListeners() function is called when the page loads. This enables it to set up the event handlers so that they are available when events fire. By checking the string returned by navigator.appVersion, the function is able to determine whether the browser is newer than Netscape Navigator 4. If so, it calls Netscape Navigator 6's addEventListener("click") method; otherwise, it calls Netscape Navigator 4's onClick event handler and sends the captured event to the appropriate function. The corresponding functions, showEventNN4Cap(evnt) or showEventNN6Cap(evnt), are each set up to capture and handle the event.

# Browser Differences

Now that you know how to capture the event object, you need to understand what you can do with it in the context of your scripts. Table 1 lists some of the most commonly used properties and methods of the three event models.

Property	Navigator 4	Navigator 6	Explorer 4+
Alt key press	modifiers	altKey	altKey
Ctrl key press	modifiers	ctrlKey	ctrlKey
Shift key press	modifiers	shiftKey	shiftKey
Cancel Bubbling	Cannot do...	preventBubble()	cancelBubble
Prevent Action (default)	return false	preventDefault()	returnValue
Next Element	Cannot do...	relatedTarget	toElement
Previous Element	Cannot do...	relatedTarget	fromElement
Keyboard key	which	keyCode	keyCode
Mouse button	which	button	button
Event Type	type	type	type
Target Object	target	target	srcElement

Table 1. Common properties of event objects.

Table 2 lists some of the most frequently used properties related to mouse coordinates.

Property	Navigator 4	Navigator 6	Explorer 4+
Element's X coordinate	(Needs to be calculated with other properties)	(Needs to be calculated with other properties)	offsetX
Element's Y coordinate	(Needs to be calculated with other properties)	(Needs to be calculated with other properties)	OffsetY
Positioned Element's X coordinate	layerX	layerX	x
Positioned Element's Y coordinate	layerY	layerY	y
X coordinate on the page	pageX	pageX	(Needs to be calculated with other properties)
Y coordinate on the page	pageY	pageY	(Needs to be calculated with other properties)
X coordinate on the window	Cannot do..	client	clientX
Y coordinate on the window	Cannot do..	clientY	ClientY
X coordinate on the screen	screenX	screenX	screenX
Y coordinate on the screen	screenY	screenY	screenY

Table 2. Coordinate properties of event objects.

## Capturing in a Compatible Fashion

Now that you are aware of the differences in how the event object is implemented across the major browser platforms, you need to know how to account for them in your scripts.

To ensure that your script can capture events regardless of the browser being used, you must include logic to determine the browser type and version, then direct the script to use the appropriate function to handle the event for that browser. Your code can make this distinction by taking advantage of the different event mechanisms that Internet Explorer and Netscape Navigator use.

*See Event*

Feb 19 2008 2008Compatibility.html Dao Dung dungdq@edt.com.vn

For production use only – not for distribution or commercial use.

8-21

Copyright © 2003 by Application Developers Training Company  
All rights reserved. Reproduction is strictly prohibited.



As you know, the Netscape Navigator browser implicitly passes the event object as a parameter to your event handling functions, while Internet Explorer simply makes the event object available via the window object's event property. The following example exploits this difference by using a simple if statement in its control-flow logic:

```
<head>
<script language="JavaScript">
// Netscape Navigator 4+
// event.propertyName
// Internet Explorer 4+
// window.event

function captureEvent(evnt) {
    var etarget;
    if(!evnt) {
        //Not passed in, must be IE
        evnt = window.event;
        //IE accesses target differently:
        etarget = evnt.srcElement;
    } else {
        //Must be Netscape!
        etarget = evnt.target;
    }
    alert("event type captured: '" + evnt.type + "' event
" +
          "target captured: '" + etarget.name + "'");
}

function setupListeners() {
    document.form1.buttoncap.onclick = captureEvent;
}
```

```
</script>
</head>
<body onload="setupListeners()">
<form name="form1">
  <input type="button" value="Capture Me"
        name="buttoncap" />
</form>
</body>
```

In the preceding example, the `setupListeners()` function is called when the document loads, and it assigns the `captureEvent()` function to the button's `onClick` event. When the button is clicked, the event fires and calls the `captureEvent()` function. This is where the control-flow logic comes into play. The function includes an `if (!evnt)` statement that determines whether the `evnt` variable references a valid event object.

If no event object was passed in, it is clear that the browser is Internet Explorer. The script assigns Internet Explorer's `window.event` reference to the `evnt` variable, and assigns the value of the event's `srcElement` property to the `etarget` variable. Otherwise, if a valid event object was passed in, the browser must be Netscape Navigator. In this case, the script is aware that the `evnt` variable must already reference a valid Netscape Navigator event object, and simply assigns its `target` property to the `etarget` variable.

Similarly, you can use this sort of control-flow logic to redirect events through your script when they need to access other properties and methods of the event object, or when they need to be handled differently based on the conditions you test for in the logic statements in your code.

# Event Types

In addition to the click event type, which has been used extensively so far, a variety of other event types are available for use in your scripts. To keep the examples relevant, this chapter focuses on Netscape Navigator 6+ and Internet Explorer 4+ event types.

**TIP:** The Netscape Navigator 4 event types are more limited in scope than those of the version 5 browsers, and Internet Explorer 3 and Netscape Navigator 3 are even more limited. Although many of the common version 5+ event types are backward compatible, please consult a reference on Netscape Navigator 4 event types to make sure that the event types you plan to use will work with the older browser versions, if compatibility is a concern.

## Mouse Event Types

The following list identifies the most common mouse events available for the version 5 browsers:

- **click:** Occurs any time the user presses a mouse button, such as with a right or left click
- **mousedown:** Occurs when the user presses a mouse button
- **mousemove:** Occurs whenever the user moves the mouse
- **mouseout:** Occurs when the user moves the mouse cursor out of a specified area or link
- **mouseover:** Occurs whenever the user moves the mouse cursor over an area or object
- **mouseup:** Occurs when the user releases the mouse button

**NOTE** For any of the events discussed in this section, the name of the corresponding event handler is the name of the event prefaced with the letters “on”.

A common use for mouse events is for changing the image *src* property to update an image when the mouse moves over it or moves off it. The mousemove type could be used to track the mouse, for example, for scripting objects to follow the mouse, such as a mousetrail.

## Keyboard Event Types: **text**, **password**, & **textarea**

The following event types are available to help your scripts deal with keyboard events:

- **keydown:** Occurs when the user presses a key
- **keypress:** Occurs when the user presses or holds down a key
- **keyup:** Occurs when the user releases a key from its depressed position

These are handy events for evaluating keyboard input while the user types, as well as for responding to modifier key-presses (**CTRL+S** or **ALT+Z** for example). If you wanted to change the screen's background color on every key press, you could achieve that easily by defining the `onkeypress` event handler of a text area.

## Loading/Unloading Event Types

- **load:** Occurs when the browser finishes loading a document or loading all the frames in a window.
- **unload:** Occurs when the user navigates away from the loaded window, document, or frameset.

Once the browser finishes unloading the current page, but before it begins loading another, the `onUnload` event fires. This is the event that generates those annoying pop-up ads you may have encountered when trying to leave a Web site! Sometimes these contain messages urging you to stay on the Web site, or to try and win you back with extra special savings. A more conscientious scripter might use the `onUnload` event handler to clean up leftover pop-up windows that opened during the application's lifespan, or alternately to alert users that their changes have not yet been saved in a critical application.

## Other Window Event Types

- **resize:** Occurs when a user or script resizes a window or frame
- **scroll:** Occurs when the user moves a scroll bar
- **error:** Fires when an error occurs (window, frameset, object)
- **focus:** Occurs when a window or frame element gains focus

- **blur:** Occurs when a window or frame element loses focus

These events are generally used for GUI enhancements. For instance, if you want to be sure a menu hangs in the top corner when the user is scrolling a window, this would be a good use for the onscroll event handler. If you wanted to trap an error, you could use the onError event handler. Or if a change of the browser window size might cause a page to display improperly, you could use the onresize event handler to capture the new measurements so that you can compensate for them in your layout.

## Form-Related Event Types

- **click:** Occurs when a form element is clicked
- **blur:** Occurs when a form element loses focus
- **focus:** Occurs when a form element gains focus
- **reset:** Fires whenever the user resets a form by clicking a Reset button
- **submit:** Fires whenever the user submits a form

Most of these event types are used for data manipulation within an HTML form. For example, on a login page you might want to return focus to the UserId text box if the user tries to leave it without entering a value. In this case, you could use UserId's onblur event handler to call a function that checks the text box's value and returns focus to the UserId text box if no value was entered. Similarly, you can use a form's onsubmit event handler to validate the fields in the form before actually submitting the page to the Web server.

# Summary

- There are three major event models: Internet Explorer 4+, Netscape Navigator 4, and Netscape Navigator 6+.
- Events propagate in two different fashions: event capturing and event bubbling.
- Event capturing starts at the outermost container, the window object, and moves down the hierarchy to the HTML element.
- The bubbling method starts at the HTML element and propagates upwards through the form and eventually on to the window object.
- Event objects contain many properties that are accessible in your scripts.
- The three major event models have different ways to access events and different properties for setting up event handlers.
- Static event objects in Netscape Navigator 4+ and Opera allow you to access various constants, such as key modifiers for **CTRL**, **ALT**, and **SHIFT** to determine whether one was held down when the event occurred. It is also used in defining the event type to capture in Netscape Navigator 4.
- In Netscape Navigator 4, event capture must be enabled before it is performed by calling `document.captureEvents([constant from the static Event object])`.
- Comment event objects are flexible in the sense that you can capture them before they reach the intended target, forward them to that target, or route them to a completely different target. Each event model has its own event properties and methods for accomplishing these goals.

# Questions

1. Which event model is W3C compliant?
2. Name a browser that starts with “O” that is mostly W3C compliant.
3. Which event handler fires when a text object loses focus?
4. If you need to script actions for when a browser is scrolled, what event type do you capture?
5. In Netscape Navigator 4+, what is the name of the object you use to specify where an event is to be directed?
6. In Internet Explorer4+, what is the equivalent of Netscape Navigator 4+’s target object?
7. In Internet Explorer 4+ what do you type to access the current event object?
8. True/False: You have to pass the event object to functions within Netscape Navigator 4+.
9. If you need to constantly catch the coordinates of the mouse, which event type must you use?
10. From within an HTML element’s event handler, you can pass an event in Netscape Navigator 4+ by using which keyword?

# Answers

1. Which event model is W3C compliant?  
**Netscape Navigator 6+ / Mozilla 1+**.
2. Name a browser that starts with ‘O’ that is mostly W3C compliant.  
**Opera**
3. Which event handler fires when a text object loses focus?  
**onBlur()**
4. If you need to script actions for when a browser is scrolled, what event type do you capture?  
**scroll**
5. In **Netscape Navigator** 4+, what is the name of the object you use to specify where an event is to be directed?  
**target**
6. In Internet Explorer4+, what is the equivalent of Netscape Navigator 4+'s target object?  
**srcElement**
7. In Internet Explorer 4+ what do you type to access the current event object?  
**window.event**
8. True/False: You have to pass the event object to functions within Netscape Navigator 4+.  
**true**
9. If you need to constantly catch the coordinates of the mouse, which event type must you use?  
**mousemove**
10. From within an HTML element's event handler, you can pass an event in Netscape Navigator 4+ by using which keyword?  
**event**

# Lab 8: Event Handling

**TIP:** Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You will find all the code in **SpreadSheet.html**, **SpreadSheet.js**, and **SpreadSheet.css**, in the same directory as the sample project. To avoid typing the code, you can cut/paste it from the source files instead.

# Lab 8 Overview

In this lab you will learn how to add simple convenience functionality into an application by incorporating events.

To complete this lab, you will need to work through two exercises:

- Mouse Interception
- Title Cell Info

Each exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

# Mouse Interception

## Objective

In this exercise, you'll create the guts of your simple mouse listener event. The application in question is the provided spreadsheet application, SpreadSheet.html and SpreadSheet.js. Feel free to browse through and see how the entire application operates.

The goal of this exercise is to create a mouseover listener on the form, so that the current cell that the mouse pointer is over reports its name to the status bar.

## Things to Consider

- You will want to use functions to separate this new functionality from the already functioning code.
- Event handlers should be initialized after the page loads, so create a function that fires with the body's onload event handler.
- The status bar is accessed using window.status.
- Remember Internet Explorer and W3C compliant browsers handle event objects differently!
- Unless it is helpful to what you are scripting outside of this class, do not worry about Netscape 4 for this exercise.

## Step-by-Step Instructions

1. Open **SpreadSheet.js** in your editor. Look over the code briefly to get an idea of what is going on. First, you will create a function to initialize the event listeners. Right under the global variables, **abc** and **formname**, create a function called **initEvents()**. You will want to use the **formname** to create a reference to the form using the eval method, with the statement **var elemnt = eval("document." + formname);**. On the next line, define the event handler, using the statement **elemnt.onmouseover = updateStatus;**.

```

var abc = "abcdefghijklmnopqrstuvwxyz";
var formname = "ssform";
function initEvents() {
    var elemnt = eval("document." + formname);
    elemnt.onmouseover = updateStatus;
}

```

2. For Netscape 6 and W3C compliant browsers, you should add an if statement with the condition (**elemnt.addEventListener**). This is known as object detection. Basically if the **if** elemnt has the method addEventListener, then this is a W3C DOM, like Mozilla 1+. On the first statement in the if block add the mouseover event listener: **element.addEventListener("mouseover", updateStatus, true);**. Finally, add one last statement to the if block, **elemnt.onmouseover = null;**, to guarantee that the new event listener you created will function in Netscape 6+ browsers (as well as in Opera).

```

function initEvents() {
    var elemnt = eval("document." + formname);
    elemnt.onmouseover = updateStatus;

    if (elemnt.addEventListener) {
        elemnt.addEventListener("mouseover",
            updateStatus, true);
        elemnt.onmouseover = null;
    }
}

```

3. Create a nested function inside of initEvents() called **updateStatus(evnt)**. You will recall from the chapter that, for browsers that implement the W3C object model, your event handling functions must include a parameter that represents the event the function will catch. Within the updateStatus()function, create a new variable called **trgt**, and add an if/else block to test the event model type for Internet Explorer compatibility, using the statement **if (window.event)**.

```
        if (elemnt.addEventListener) {
            elemnt.addEventListener("mouseover",
                updateStatus, true);
            elemnt.addEventListener("click", setTitle, true);
            elemnt.onmouseover = null;
            elemnt.onclick = null;
        }

        function updateStatus(evnt) {
            var trgt;
            if(window.event) {
            } else {
            }
        }
    }
```

4. Within the if block, assign **window.event** to the **evnt** variable to make sure **evnt** is defined properly if the client browser is Internet Explorer. On the next line, set **trgt** to reference the event's target, using the statement **trgt = evnt.srcElement;**. If the client's browser is not Internet Explorer, then the else block will execute, and **trgt** should be set to reference **evnt.target**.

```
function updateStatus(evnt) {
    var trgt;
    if(window.event) {
        evnt = window.event;
        trgt = evnt.srcElement;
    } else {
        trgt = evnt.target;
    }
}
```

5. Create an if block with the condition **trgt.type == "text" && trgt.name.match(/cell/g)**. This ensures that you are in a textfield within the spreadsheet, which has "cell" within the name.

```
function updateStatus(evnt) {  
    var trgt;  
    if(window.event) {  
        evnt = window.event;  
        trgt = evnt.srcElement;  
    } else {  
        trgt = evnt.target;  
    }  
    if (trgt.type == "text" &&  
        trgt.name.match(/cell/g)) {  
    }  
}
```

6. Finally, inside of the if /else statement, populate the status bar with the target's name using the statement **window.status = trgt.name;**

```
if (trgt.type == "text" &&  
    trgt.name.match(/cell/g)) {  
    window.status = trgt.name;  
}  
}
```

7. Test the application by running **SpreadSheet.html** in your browser. When you mouseover a cell, its name should appear in the status bar, as shown in Figure 7.

**Lab 8:**  
**Event Handling**

---

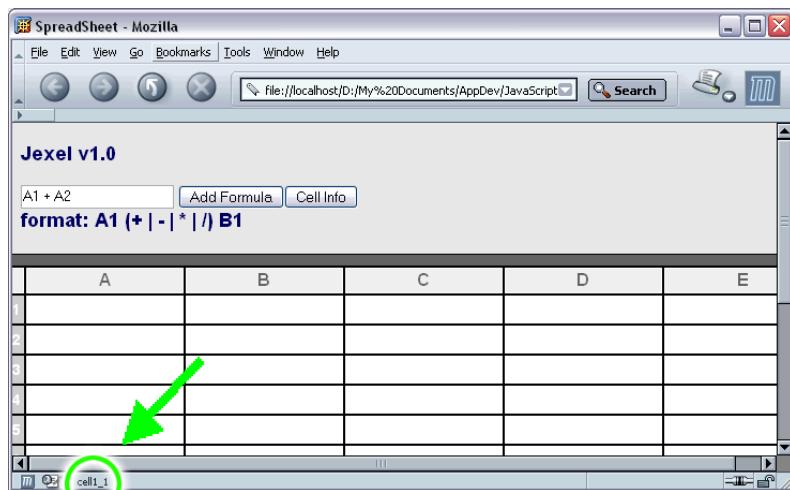


Figure 7. Cell 1\_1 shows in the status bar in Mozilla.

# Title Cell Info

## Objective

In this exercise, you will make the title show the information provided in the cell on the window title area. Capturing the onclick event, the title will be adjusted to inform the user of the page they are on, and the current cell's information.

## Things to Consider

- With text, password, and textarea objects, the onclick event bubbles up through the document hierarchy, or captures downwards through the hierarchy. However, onBlur and onChange only fire at the object level.
- Prior to Internet Explorer 4 and Netscape Navigator 6, you can only read the document title.
- The cell's onclick event will still fire, even when you catch it higher up in the document.
- You will need to use a provided method, getNumAlpha(cell column number) to determine what letter(s) the current column is. (A – ZZZZ and beyond).

## Step-by-Step Instructions

- This exercise builds on the functionality of the previous exercise. While the contents of updateStatus are similar to the function needed for this exercise, to reduce confusion, updateStatus will be copied to a new function called setTitle. Remove the line that sets the window.status.

```
function setTitle(evnt) {  
    var trgt;  
    if(window.event) {  
        evnt = window.event;  
        trgt = evnt.srcElement;  
    } else {  
        trgt = evnt.target;  
    }  
}
```

*Lab 8:*  
*Event Handling*

---

```
if (trgt.type == "text" &&
    trgt.name.match(/cell/g)) {
    window.status = trgt.name;
}
}
```

2. Inside of the second if/else block, the one that checks for the cell type and name, start a new string: **var strTitle = “SpreadSheet: ”** and then separate the properties located in the target cell’s name, using the statement **var id = trgt.name.substring(4, (trgt.name.length));**.

```
if (trgt.type == "text" &&
    trgt.name.match(/cell/g)) {
    var strTitle = "SpreadSheet: ";
    var id = trgt.name.substring(4,
        (trgt.name.length));
}
}
```

3. Redefine your code so that the variable **id** refers to an array of cell information by splitting itself into its component elements, using the statement **id = id.split(“\_”);**. This places the column number in id[0] and the row number in id[1].

```
if (trgt.type == "text" &&
    trgt.name.match(/cell/g)) {
    var strTitle = "SpreadSheet: ";
    var id = trgt.name.substring(4,
        (trgt.name.length));
    id = id.split("_");
}
}
```

4. Append cell information onto the strTitle with the statement **strTitle += “ Current Cell: ” + getNumAlpha(id[0]) + id[1];**. The function **getNumAlpha()** returns the column letter equivalent of the column number you pass in.

```

if (trgt.type == "text" &&
    trgt.name.match(/cell/g)) {
    var strTitle = "SpreadSheet: ";
    var id = trgt.name.substring(4,
        (trgt.name.length));
    id = id.split("_");
    strTitle += " Current Cell: " +
    getNumAlpha(id[0]) + id[1];
}

```

5. Add an if statement to the block with the condition (**trgt.value**). For the first statement in the new if block, add the value of the cell to the title string: **strTitle += “ – Value: ” + trgt.value;**

```

id = id.split("_");
strTitle += " Current Cell: " +
getNumAlpha(id[0]) + id[1];
if(trgt.value) {
    strTitle += " -- Value: " + trgt.value;
}

```

6. At the end of the if block, add another if statement with the condition (**trgt.formula**), then for the first line in that block add the formula information to the title: **strTitle += “ – Formula: (” + trgt.formula + “)”;**

```

if(trgt.value) {
    strTitle += " -- Value: " + trgt.value;
}
if(trgt.formula) {
    strTitle += " -- Formula: (" +
    trgt.formula + ")";
}

```

7. Finally, add a line immediately following the if block to change the window title to reflect the new title string, using the statement **document.title = strTitle;**

```
        if(trgt.formula) {  
            strTitle += " -- Formula: (" +  
                trgt.formula + ")";  
        }  
document.title = strTitle;
```

8. Now that the title function is complete, you must register the events so that they can execute. Proceed to the top of the **initEvents()** function, and add a new **elemnt.onclick** definition under the **elemnt.onmouseover** line, which was defined in the first exercise, that points to the **setTitle()** function.

```
elemnt.onmouseover = updateStatus;  
elemnt.onclick = setTitle;
```

9. Two additional lines must be added to the if statement on the next line. First, for W3C browsers, add an **elemnt.addEventListener()** for the “click” action, passing in **setTitle**, and **true** as parameters. On the next line, set **elemnt.onclick** to null, since the listener is already defined.

```
elemnt.onclick = setTitle;  
  
if (elemnt.addEventListener) {  
    elemnt.addEventListener("mouseover",  
        updateStatus, true);  
    elemnt.onmouseover = null;  
    elemnt.addEventListener("click", setTitle, true);  
    elemnt.onclick = null;  
}
```

- NOTE** Steps 8 and 9 have already been added to the completed lab file. However, they have been commented out so that first exercise can be viewed. To launch this exercise, be sure to uncomment the lines described above.

10. Now it is time to test the application. When you mouseover a cell, its name will appear in the status bar like before. However, if you click on a cell, the title information will be displayed in the title bar of the window, as shown in Figure 8.

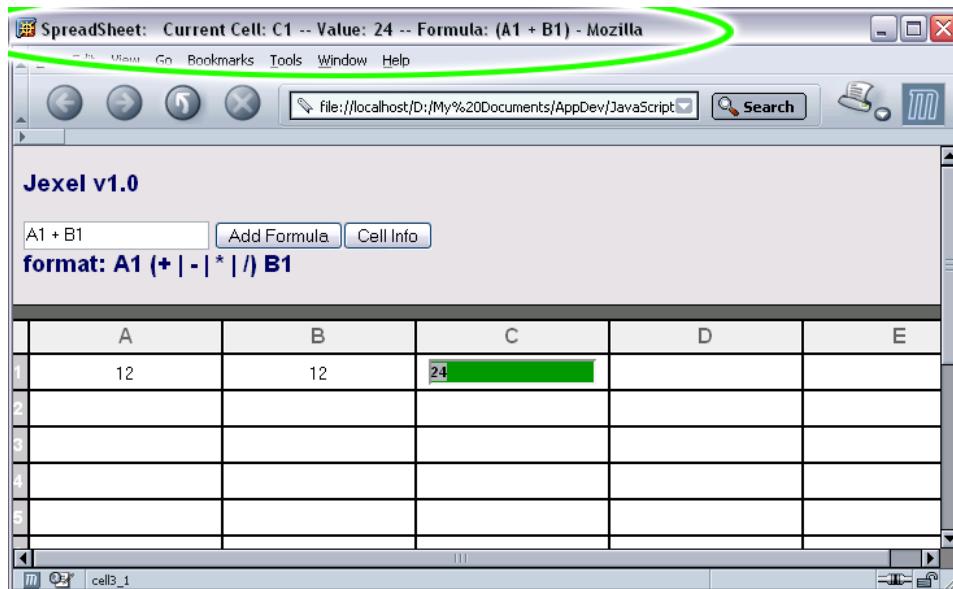


Figure 8. Mouse clicked on cell C1, with the formula in the Mozilla browser.

*Lab 8:  
Event Handling*

---

Feb 19 2008 3:29PM Dao Dung dungdq@edt.com.vn  
For production evaluation only – not for distribution or commercial use. **JavaScript 1.5**

Copyright © 2003 by Application Developers Training Company  
All rights reserved. Reproduction is strictly prohibited.



# Error Handling

## Objectives

- Read error messages effectively.
- Investigate problems with your scripts.
- Learn about exception handling.
- Learn to create try...catch blocks.
- Create your own custom Error objects.
- Discover a free JavaScript debugger.
- Learn how to effectively test your scripts.

# Reading Error Messages

Even the best programmers and scripters make mistakes in their code. What separates them from mediocre developers is the ability to effectively test and debug their work. The first step to fixing an error in your script is, of course, being aware of the error. At first, this might seem like a fairly simple thing to do, but it is actually far more complex than it seems.

Most modern browsers do their best to suppress obvious error notification, which makes it difficult to find scripting errors. The early scriptable browsers displayed an obvious error dialog box, but this proved too disruptive for users, especially nontechnical users who often thought that they were to blame for breaking the page. Suppressing these dialog boxes in later browser versions has been good for users, but makes it far too easy for scripters to overlook errors in their code.

## Displaying Errors

Each of the major browsers has a different way of displaying errors to the user. In Internet Explorer 5+, error notifications are displayed in the browser window's status bar. Figure 1 shows an example.

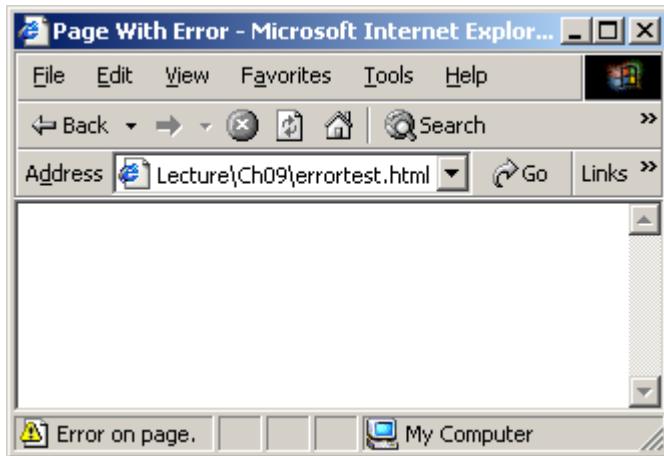


Figure 1. Internet Explorer notifies the user of script errors in the left edge of the status bar.

If you double-click on the icon in the status bar, the error dialog box appears. Click on the Show Details button to see useful details about the error, including the line number within your script file on which the error occurred. Figure 2 shows the Internet Explorer error dialog box, expanded to show details.

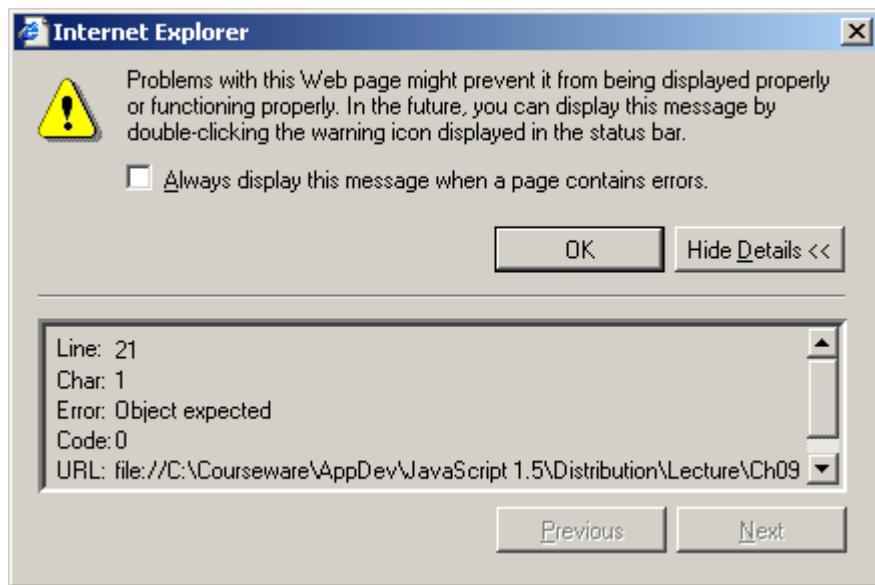


Figure 2. The Internet Explorer error dialog box.

It is advisable to click the check box labeled “Always display this message when a page contains errors” and leave it checked. This choice displays the Internet Explorer error dialog box every time the browser encounters an error in your scripts, so that you don’t have to monitor the status bar continually. You can also enable this option by going to **Tools|Internet Options|Advanced|Browsing** and checking the check box labeled “Display a notification about every script error.”

To view errors in a Netscape-based browser, open the JavaScript console by selecting **Tools|Web Development|JavaScript Console** from the main menu. This displays a detailed list of all of the errors within a script. Figure 3 shows the JavaScript console, which lists several errors that it found in a script.

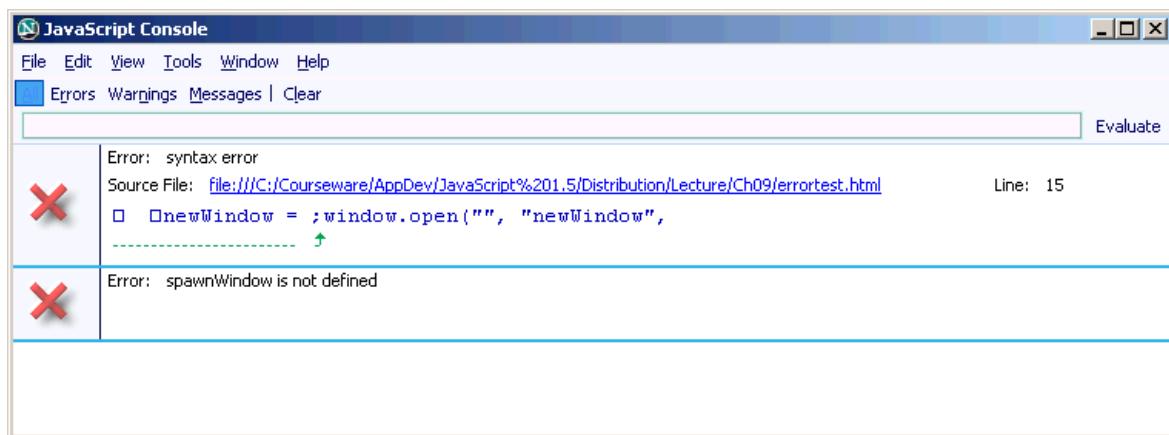


Figure 3. The JavaScript console displays a detailed list of all script errors.

You can leave the JavaScript console open as you test your pages, and subsequent error messages will be appended to the list.

**TIP:** If you leave the Netscape Navigator JavaScript console open as you test, be sure to clear it before each test. Otherwise, you may confuse old error messages that you have already fixed with current errors.

When tackling issues listed in the JavaScript console, you should always start by addressing the first error in the list. Errors appear in the list in the order that they occurred, and quite often errors that appear later in the list were caused by the earlier errors. If you test your script after fixing the first error, you may be pleasantly surprised to find that the fix also addresses many subsequent errors.

## Investigating Problems

The exact information within error messages differs depending on which browser you are using, but generally the messages include three pieces of information: the name of the file in which the error occurred, the location of the error within that file, and a description of the error. Using these bits of information, you should be able to track down most errors in your source code.

Before you try to track down an error, you should be aware of a difference in how the major browsers report the name of the offending file. When an error occurs in a script that is embedded within an HTML page, both Internet Explorer and Netscape Navigator properly list the name of that HTML page in their error messages. However, if the error occurs in a separate .js file that an HTML page references, the filename reported in the error message will differ depending on the browser. Netscape Navigator's error message properly reports that the error occurred in the external .js file. However, Internet Explorer's error message reports that the error occurred in the HTML page that references the .js file, which makes it a bit harder to track down the error unless you are aware of this quirk in Internet Explorer.

Generally, the error line number in the report reflects the exact line on which a script error occurs. However, on some occasions Internet Explorer misinterprets the context of errors; it can miss closing braces and report the error on a later line than it truly is. This usually occurs when the missing brace is followed by a new function declaration: Internet Explorer thinks that the new function should be nested within the function with the missing brace. In addition, if an error arises in a script that was referenced from an HTML event handler, such as onClick, Internet Explorer will commonly report that the script error originated in the line of HTML where the event handler is defined, rather than in the offending line of JavaScript code. If you are aware of Internet Explorer's error reporting discrepancies, it will help you identify the location of the real error.

The details of the error report also differ based upon the browser that loads the page. While the detail text is meant to help you understand the nature of the error, the text can be quite cryptic. Complete coverage of even the most common error messages and their meanings is beyond the scope of this text due to the number of possible errors and how the different browsers interpret them. In time, though, you will become familiar with them.

# Exception Handling

In programming terminology, an exception can be defined as an occurrence that falls outside the range of expected results. This is a kind way of saying that something went wrong while your code was running. Exception handling involves making provisions in your code to allow for the unexpected.

Generally speaking, scripters make little distinction between an exception and an error—an exception causes a script to deviate from the intended execution in unexpected ways, typically resulting in an error. In discussions of these JavaScript concepts, it is perfectly acceptable to think of errors and exceptions as the same thing.

Whenever an exception is raised, the browser creates an Error object that contains information about the exception. The browser uses these Error objects to provide you with the error messages mentioned in the previous sections. The real utility of these Error objects, however, is as a mechanism for handling these exceptions intelligently.

## Try...Catch...Finally Blocks

One method for handling exceptions is to surround the code with a try...catch...finally block, to encapsulate any script statements that may generate a run-time error. When you use this mechanism, your scripts can handle any possible exceptions that arise within those statements in a controlled manner and can prevent any resulting errors from appearing in the browser's error reporting mechanism. The syntax for a try...catch...finally block is as follows:

```
try {  
    // statements that might generate an error  
}  
catch (errorVariable) {  
    // code that handles any errors that arise  
}  
finally {  
    // statements to execute, regardless of error status  
}
```

In the try block, place the statements that you think could potentially raise an exception. If an exception is raised on any line of code in the try block, control immediately passes to the catch block, and the statements within it are

executed. In the finally block, you can place statements that will run regardless of whether an exception was raised. The finally block is optional, and is generally used to close any resources that were still open when the error occurred, such as access to a file or a connection to a database.

Try...catch...finally blocks can also be nested, but each try block must have a corresponding catch and/or finally block.

```
try {  
    // outer try block statements  
    try {  
        // inner try block statements  
    }  
    catch (innerError) {  
        // inner catch block statements  
    }  
}  
catch (outerError) {  
    // outer catch block statements  
}  
finally {  
    // finally block statements  
}
```

See  
***RandomError.html***

Notice that the catch keyword is followed by a set of parentheses that contain a variable name. When an exception occurs inside a try block, the browser creates an Error object and passes it as a parameter to the catch block. You can reference the Error object from within the catch block via the variable name, to access error information stored in the object's properties and determine how to handle the error based on this information. The following example purposely generates one of three random errors in order to illustrate this point:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
  "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="en" lang="en">
<head>
  <title>Generate a Random Error</title>

  <script language="JavaScript" type="text/javascript">
    function generateRandomError(randErrorNum) {
      try {
        //This raises a SyntaxError.
        if (errorNum == 1) {
          eval("var num = 1 +; 7");
        }
        //This also raises a SyntaxError.
        else if (errorNum == 2) {
          eval("var num =;" );
        }
        //This raises a ReferenceError.
        else if (errorNum == 3) {
          eval("var num = 1 + a;");
        }
      }
      catch (err) {
        switch (err.name) {
          case "SyntaxError":
            alert("A syntax error was raised!");
            break;
          case "RangeError":
            alert("A range error was raised!");
            break;
          default:
            alert("An unexpected error of type " +
              err.name + " was raised!");
        }
      }
    }
  </script>
</head>
<body>
  <h1>Generate a Random Error</h1>
  <p>Select a number from the dropdown menu below, then click the button.</p>
  <form>
    <label>Error Type:</label>
    <select id="errorType">
      <option value="1">Syntax Error</option>
      <option value="2">Range Error</option>
      <option value="3">Reference Error</option>
    </select>
    <input type="button" value="Generate Error" onclick="generateRandomError(errorType.value)" />
  </form>
</body>
</html>
```

Feb 19 2008 3:29PM Dao Dung dungdq@edt.com.vn  
For production evaluation only – not for distribution or commercial use. **JavaScript 1.5**

```
        }
    }
</script>

</head>
<body onload="generateRandomError(
    Math.ceil(Math.random() * 3))">
</body>
</html>
```

The onload event in the preceding example generates a random number between one and three. This number is used to determine which of three possible evaluations will be performed in the try block; each evaluation will raise an exception. Two of the evaluations result in an Error object that has the name property SyntaxError, while the third evaluation results in an Error object that has the name property ReferenceError.

In the preceding example, the catch block code anticipates the possible SyntaxError and handles it gracefully via a switch...case statement that checks the name property of the Error object and returns an alert to the user that specifies the error type. The example also includes a case statement with code to handle a RangeError, but does not anticipate that a ReferenceError will occur. That's okay though, as the default element of the switch...case construct handles any unexpected errors and returns an appropriate alert to the user. This example is a little contrived for real-world use, but it aptly illustrates how you can use the properties of the Error object to provide specialized error handling in your catch blocks.

## Custom Error Objects

In addition to handling exceptions that violate the rules of the language, JavaScript's exception handling model enables you to create custom Error objects that help enforce business rules in your scripts. For instance, if a text field on a page should allow the user to input only a certain range of characters, you can throw a custom exception object when the user enters any invalid characters. The syntax for throwing an exception is as follows:

```
throw value
```

The value thrown can be of any data type, so it is possible to throw an error whose value is a string, which your catch block can evaluate and act upon. If your business rules dictate that a certain variable's value must be less than 10, your script can evaluate that variable in a try block and throw an exception if the value violates the rule, as shown in the following code snippet:

```
try {
    if (x >= 10) {
        throw "The number entered must be less than 10";
    }
} catch (errorVariable) {
    alert(errorVariable);
}
```

Since `errorVariable` is a String, it can be used by the `alert()` function to display the message for the user.

There is a potential problem with the example above, however. If the code in the try block throws a browser-based exception, the value passed to the catch block will be an Error object rather than a string. Since the current catch block code is not equipped to deal with an Error object, it will mask a potentially serious error without handling it properly.

For this reason, it is good practice to throw exceptions that only pass Error objects, so that the code in the catch block can handle exceptions consistently. To ensure that your code only throws Error objects, you will need to create a custom Error object that is capable of describing your exception. To create a custom Error object, you would use the following syntax:

```
var errorObjName = new Error(message); // custom Error

throw errorObjName; // throw custom Error object
```

You can then use the variable name that was thrown to access the Error object's properties and methods.

**NOTE** The Error object has several properties, but the only ones that both Internet Explorer and Netscape Navigator support uniformly are `Error.name` and `Error.message`.

*See Custom  
Exceptions.html*

The following example illustrates how to create your own Error objects and throw exceptions. The example creates a page that has a label, an input box, and a button. The purpose of the page is to allow the user to enter some input into the text box and click the button. The onClick event of the button calls a function to validate the input and checks whether it adheres to the rule stated in the label.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="en" lang="en">
<head>
    <title>Input Validator</title>

    <script language="JavaScript" type="text/javascript">
        function getErrorObj (msg) {
            var e = new Error(msg);
            e.name = "CUSTOM_ERROR";
            return e;
        }

        function validateInput(num) {
            try {
                if (isNaN(num)) {
                    throw getErrorObj("Your input was " +
                        "not a number");
                }
                else if (num < 1) {
                    throw getErrorObj("The number must " +
                        "be greater than 1");
                }
                else if (num > 10) {
                    throw getErrorObj("The number must " +
                        "be less than 10");
                }
            }
            else {
```

```
        alert("The input is valid");
    }
}
catch (err) {
    if (err.name == "CUSTOM_ERROR") {
        alert(err.message);
    }
    else {
        alert("The browser threw an error of " +
            "type: " + err.name +
            ". Description: " + err.message);
    }
}
</script>

</head>
<body>
<form>
    Enter a number between 1 and 10:
    <input type="text" name="numField">
    <input type="button" value="Validate"
        onClick=
            "validateInput(this.form.numField.value)">
</form>
</body>
</html>
```

In the preceding example, notice that the validateInput() function checks the user input to make sure that it is a number, and that its value falls within the target range. If the input fails either of these tests, then the validateInput() function throws an exception that is created via a call to the getErrorObj() function.

Since the only difference in the three possible errors is the messages they generate, getErrorObj() is defined as a separate function to handle the common tasks involved in generating the Error object. Its job is to create a new custom Error object containing the specific error message that was passed in as a parameter, assign the Error object a name, and return it to the calling statement.

Once the exception is thrown, execution continues in the catch block, which will perform a different error handling routine depending on whether the Error object is a custom object or one created by a browser exception.

# JavaScript Debugging

As stated earlier in the chapter, even the best developers have errors in their code. It is important to learn how to track down and identify these errors in an organized, straightforward way, to help minimize their impact on the project.

Often, entry-level scripters resort to littering their code with alert lines and document.write() methods to report the value of any variables they are trying to track at runtime. While this method may serve the purpose, it requires a significant amount of time to create test blocks, insert test lines, and then go back and remove them from the code before putting it into production.

For some time there was a notable lack of tools to automate the process of debugging JavaScript code. Netscape and Mozilla enabled you to see any errors that arose via the JavaScript console, but more advanced debugging functionality was simply unavailable. Developers with a sufficient budget could purchase a Microsoft-centric commercial IDE called Visual Studio, which includes a tool called Visual InterDev for debugging JavaScript or JScript within Internet Explorer. However, Visual Studio includes many other tools for development in a variety of languages, and the cost of the package makes it prohibitive to purchase solely for JavaScript debugging.

Fortunately, tools are now available to let you evaluate your code at runtime and help you with the debugging process.

## The Venkman Debugger

Recently, the Mozilla project incorporated a free debugging tool with its distribution, called the Venkman debugger. Venkman is included at no charge with any version of Mozilla released after October 3, 2002, and aims to provide IDE-quality debugging tools to JavaScript developers who are working in the Mozilla or Netscape environment. One of the nice features of this debugger is that it is cross-platform compatible: any machine that can run the Mozilla or Netscape browser can also run Venkman.

**NOTE** The version of Mozilla that is included with this course contains the Venkman debugger. If you are using a different version of Mozilla, you can check the release date by clicking **Help|About Mozilla** on the menu.

Within the Venkman environment, you can take advantage of advanced debugging features, such as setting breakpoints in your code. If you add a breakpoint on a line of code after a variable has been declared, and then run your script, the debugger will execute the script until it reaches the breakpoint. The debugger will then pause the program and allow you to see the value of

that variable at that point. You can then step through your code line by line and see how the state of the application changes at any given point during your script's execution. This is helpful because it maintains a separate environment for debugging and avoids the time and tedium associated with using test code snippets to determine runtime values.

Figure 4 displays what the Venkman debugger looks like when it stops for a breakpoint:

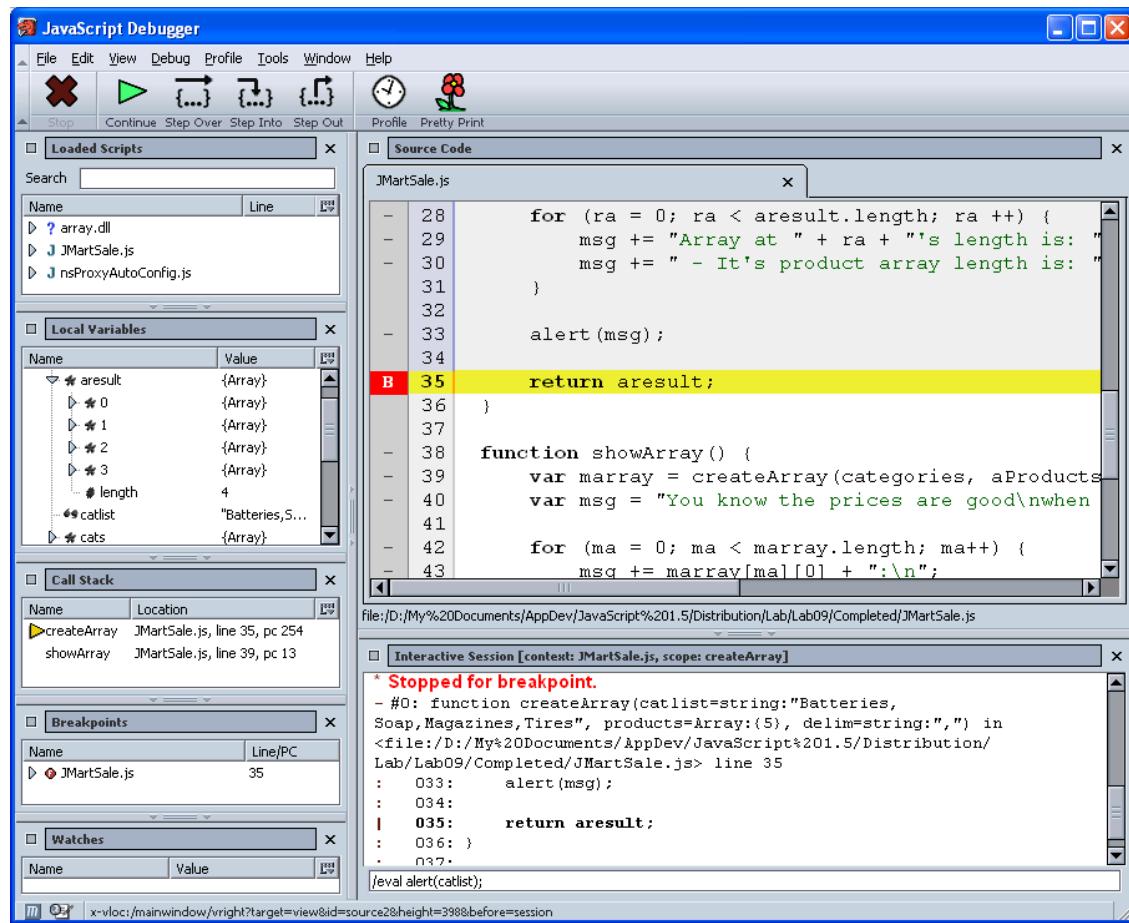


Figure 4. The Venkman debugger.

You can launch the Venkman debugger in Mozilla or Netscape by choosing **Tools|Web Development|JavaScript Debugger** from the browser's menu. If Venkman is not included in your version of Netscape or Mozilla, you might have to upgrade to be able to work with it. More information about Venkman is available from:

<http://www.mozilla.org/projects/venkman>

## The Venkman Interface

This section provides a basic overview of the Venkman debugger's user interface and a brief discussion of how to use it to test your scripts. Debugging is a complex topic that blends art and science. While comprehensive coverage is beyond the scope of an introductory JavaScript course, it will help you to learn good debugging skills early on.

See  
[DuplicateVariable.html](#)

Bugs can occur in applications for any number of reasons. For the following example, a like-named variable is the culprit:

```
<head>
<script language="JavaScript">
    function returnAutomobile() {
        var thecar = "Buick";
        var thecars = "a popular 80's band.";
        ...
        return thecars;
    }
</script>
</head>
<body>
    <form>
        <input type="button" name="show" value="Show Auto"
               onclick="alert('Car: " + returnAutomobile())"/>
    </form>
</body>
```

In the preceding example, the scripter confused the variable that the code should have returned with a similarly named variable. While this is a relatively trivial example, it might be much more difficult to track down an error like this in a complex, thousand-line JavaScript program. Note that this would not result in a browser error, but is simply the result of an error in the code's logic.

When the script generates an alert with unexpected output, you would probably deduce that the variable is being set incorrectly, and realize that you must search through the code to find the source of the error. Rather than go spelunking randomly through the script, this would be an ideal time to use the Venkman debugger.

## Loading the Script

Once the page containing the JavaScript code is loaded into the Mozilla browser, the first thing you should do is get Venkman started by selecting **Tools|Web Development|JavaScript Debugger** from the browser's menu. Once Venkman is running, you can choose which script to evaluate by looking in the loaded scripts panel, shown in Figure 5.

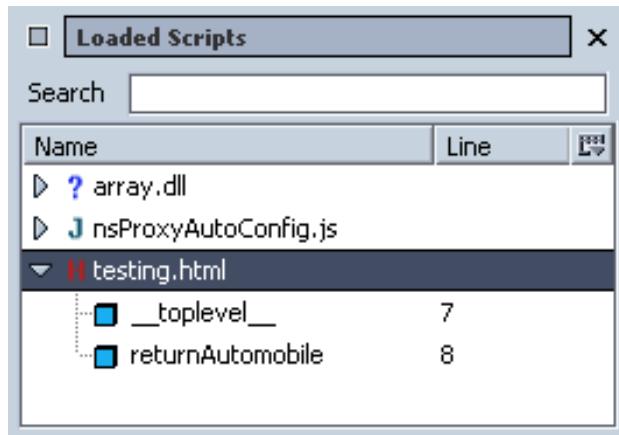


Figure 5. The Venkman Loaded Scripts panel.

Double-click the script in question to load the source into Venkman's source code panel. You will also notice in Figure 5 that you can double-click on the `returnAutomobile()` function listing, which causes the debugger to jump right to the point where that function is defined in the source code panel. In the code panel, you can peruse the code to try to identify the problem, or you can try to pin down the source of the error by setting breakpoints, as you will see later in the section.

## Local Variables View

The Local Variables view, shown in Figure 6, displays the current variables in the running code. The *scope* heading lists any variables that are in the scope of the current function; the *this* heading lists any variables that available for the whole window.

Local Variables	
Name	Value
scope	{Call}
thecar	"car"
thecars	"a popular 80's band."
this	{Window}

Figure 6. The Venkman Local Variables view.

The variable names appear in the Name column, and their corresponding values are in the Value column. Note that longer string values may simply display as a character count rather than the string itself. Arrays have an arrow next to them, which you can click to expand the array and display the values at each index position. If you double-click on a value in this view, Venkman will present you with a prompt asking whether you want to insert a new value for that variable. This enables you to change the value manually while the script is running, which can be particularly handy when you need to test a range of values while debugging your script.

## Debugging Toolbar

The Venkman debugging toolbar is shown in Figure 7. The Stop button at the far left of the toolbar pauses the page that is currently executing in Mozilla. The Continue button causes your script to continue its normal execution after either the stop button or a breakpoint has paused the page.



Figure 7. The Venkman toolbar.

## Step Over, Step Into, and Step Out

Once the debugger has paused, the buttons Step Over, Step Into, and Step Out enable you to resume execution and control the level of granularity with which the debugger processes your code.

The Step Over button executes all the lines in the current function at once, then proceeds to the next function call in the program. If there are no additional function calls, the application simply resumes normal execution in Mozilla.

The Step Into button executes the line of code at the current position, then proceeds to the next line in the script, and continues executing line by line. If

the current line of code happens to call a function, Step Into will move into the code block for that function and execute it one line at a time. When that function returns, Step Into will return control to the line that called the function and continue executing each line in sequence from that point.

*See [Basic Venkman.html](#)*

To illustrate how these functions can be used to help you debug your code, the following example code contains some logic errors. Open the file in Mozilla, then launch the Venkman debugger and load the file called BasicVenkman.html from the Loaded Scripts panel. Set a breakpoint on the first statement in the showCars() function, as indicated in the comment line above it. Then, return to the Mozilla browser and click the Show Auto button on the HTML page:

```
<head>
<script language="JavaScript">

    var thecar = "Buick";
    var thecars = "a popular 80's band.";

    function showCars() {
        // Set BREAKPOINT on next line
        alert("Car: " + returnAutomobile());
        alert("Band: " + returnBand())
    }

    function returnAutomobile() {
        var automobile = thecars;
        return automobile;
    }

    function returnBand() {
        var band = thecar;
        return band;
    }
}
```

```
</script>
</head>
<body>
<form>
<input type="button" name="show" value="Show Auto"
       onclick="javascript: showCars()"/>
</form>
</body>
```

The Venkman debugger will pause execution when it encounters the breakpoint in the showCars() function and wait for your input before continuing. If you click the Step Into button, the debugger will forward control to the returnAutomobile() function and execute it line by line.

Continue clicking the Step Into button until control passes to the return statement for returnAutomobile(). A quick glance at the Local Variables panel will tell you that something is amiss: “a popular 80’s band” does not seem like a valid value for the automobile variable. Another glance at the Source Code panel shows you where this assignment took place in the code, making it easy to find and fix the problem. Clicking the Step Into button twice more will return control to the showCars() function and trigger the alert dialog box, which displays the erroneous data.

You can continue clicking the Step Into button to execute each line in the script in sequence, or click the Step Over button to let the subsequent lines execute in the background. Alternately, if you are inside the code block of a particular function and would like the remainder of its statements to execute all at once, you can simply click the Step Out button.

Both Step Out and Step Over cause the debugger to pause execution when it encounters the next function call or continue processing the script to completion if no subsequent function calls exist. If you would simply prefer to continue normal script execution without any further pauses, you can click the Continue button.

**NOTE** Venkman is not without some minor quirks, and you may encounter one the first time you use the Step Into function to step through the BasicVenkman.html script. On the first pass, you may notice that Step Into bypasses the returnAutomobile() function and only steps through the call to returnBand(). If you try running the program again, however, you should see that Step Into works as expected and executes each function line by line.

## Profile and Pretty Print

The Profile button activates Mozilla's profiling feature, which starts recording usage information for the current page in Mozilla. The information includes current active scripts, how many times the functions were called, how much time they take to execute, and the line numbers in which that function is defined. You can access this data by choosing Profile from the main menu, then saving the profile data to a text file. The Profile button is an on/off toggle switch, and you will know when it is activated because a green checkmark appears on the button.

The functionality of the Pretty Print button is fairly simple. Pretty Print changes the Source Code panel to show only the current active function, rather than the entire JavaScript block that appears within the <script> tags. This helps to reduce onscreen clutter, and lets you focus on the specific segment of the code that you are debugging. The Pretty Print button is also an on/off toggle switch and displays a green checkmark on the button when it is activated.

## Breakpoints Panel

The Breakpoints panel, as displayed in Figure 8, shows you which documents and functions currently contain breakpoints. This is handy for identifying all the breakpoints that you need to remove after you find and correct your code errors.

Name	Line/PC
JMartSale.js	35
testing.html	12
returnAutomobile	[14]

Figure 8. The Venkman Breakpoints panel.

## Call Stack Panel

The Call Stack panel lists all the functions that are currently active on the page. If you double-click on a function name in the Call Stack panel, Venkman will bring up that function in the Source Code panel. In addition, if you right-click on a function, you can selectively turn debugging on and off for that particular function. Figure 9 shows the Call Stack panel.

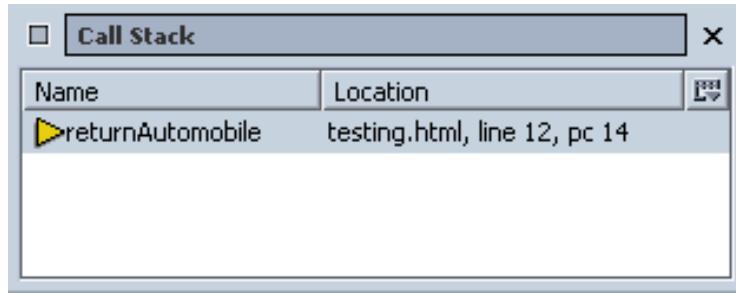


Figure 9. The Venkman Call Stack panel.

## Interactive Session Panel

The Interactive Session panel displays some of the inner workings of the debugging session. In this view, you can see the line of code that the Venkman debugger paused on, and call additional commands to aid in the debugging process. The Interactive Session panel is shown in Figure 10. While it is beyond the scope of this course to cover all of the available commands, you can get started with a few basics.

A screenshot of the Venkman Interactive Session panel. The window title is "Interactive Session [context: testing.html, scope: returnAutomobile]". The main area shows a stack trace and some command history. The stack trace includes lines 35 through 40 of a script, with line 35 being the current breakpoint. Lines 35 and 36 show code related to alerting a message. Lines 37 through 40 show the return of a result. Below the stack trace, there are two messages: "\* Continuing from breakpoint." in green and "\* Stopped for breakpoint." in red. The command line at the bottom shows the command "/eval alert(catlist);".

```
- #0: function createArray(catlist="Batteries,Soap,Magazines,Tires", products=Array:[5], delim=string:",") in <file:/D:/My%20Documents/AppDev/JavaScript%201.5/Distribution/Lab/Lab09/Completed/JMartSale.js> line 35
:   033:     alert(msg);
:   034:
| 035:     return arestult;
:   036: }
:   037:
* Continuing from breakpoint.
* Stopped for breakpoint.
- #0: function returnAutomobile() in <file:/D:/My%20Documents/AppDev/JavaScript%201.5/Distribution/Lab/Lab09/Completed/testing.html> line 12
:   010:     var thecars = "a popular 80's band.";
:   011:
| 012:     return thecars;
:   013:   }
:   014: </script>

/eval alert(catlist);
```

Figure 10. The Venkman Interactive Session panel.

The most useful command for the Interactive Session is /help, which opens a Web page with a list of all the commands available. The next most useful command is /eval. In Figure 10 you can see that /eval alert(catlist); is typed in the command line. This command calls alert(catlist); from the script you are debugging, and displays the results live. So, for example, if a string listed in the Local Variables panel is too long to display there, then you can type '/eval alert(longstringname)' in the Interactive command line, and it will display the alert through the script in Mozilla.

## Testing Code

You cannot actually edit the code within the Venkman debugger, so it would be wise to have your code open in a separate editor—you can change the source code, refresh the page that contains it in Mozilla, and then debug it in Venkman.

Occasionally, you may have problems getting breakpoints to code execution in Venkman. This usually occurs when the source code you are viewing in the debugger has changed in an external editor, rendering the two versions out of synch. To resolve this problem, simply close the current tab in Venkman's Source Code panel, refresh the Web page that you are debugging in Mozilla, and continue.

You might have noticed that if you click on the breakpoint indicator of a line, it turns to an orange f symbol. Clicking the f will clear the breakpoint completely. The orange f stands for future breakpoint and means that if the file is loaded in the future with the same line of code or character makeup, that breakpoint will automatically be set for the line. This is helpful when you are working on a large project, or you know you will have to return to this code at some point and perform some additional debugging.

# Summary

- When testing your scripts, you should enable error notification for your browser.
- Error notifications can be helpful, but aren't always accurate in pointing to the offending script statement.
- Write your scripts to handle unexpected interaction.
- Enclose code that is prone to exceptions in try...catch...finally blocks.
- The browser creates an Error object when an exception is raised; the object's properties can be referenced in your catch blocks.
- Create custom Error objects to improve handling of your business rules.
- The Venkman debugging tool is a full-featured JavaScript debugger.
- To execute each line in your JavaScript code sequentially, use the Step Into feature of Venkman.
- To place a breakpoint in Venkman, click on the line of script you want to stop at when the JavaScript executes.

# Questions

1. When tackling errors listed in the Netscape Navigator JavaScript console, should you start with the first error listed or the last?
2. What three pieces of information are in error messages that the browser returns?
3. True/False: The line number listed in an error message always reflects the line that the error occurred on.
4. What control structure is used to handle exceptions?
5. True/False: When an exception is thrown, the exception can be of any type.
6. What two Error object properties have common support among all browsers?

# Answers

- When tackling errors listed in the Netscape Navigator JavaScript console, should you start with the first error listed or the last?

**Always start with the first error in the list, because errors listed later may be due to the interpreter being thrown off by the first.**

- What three pieces of information are in error messages that the browser returns?

**The filename that the error occurred in, the line number that the error occurred on, and a text description of the error**

- True/False: The line number listed in an error message always reflects the line that the error occurred on.

**False. In several instances, the browser will not be able to correctly report the line number that the error occurred on.**

- What control structure is used to handle exceptions?

**A try...catch...finally block**

- True/False: When an exception is thrown, the exception can be of any type.

**True. The value accompanying a throw statement can be of any type.**

- What two Error object properties have common support among all browsers?

**Error.name and Error.message**

# Lab 9: Error Handling

**TIP:** Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You will find all the code in **JMartSale.html** and **JMartSale.js**, in the same directory as the sample project. To avoid typing the code, you can cut/paste it from the source files instead.

# Lab 9 Overview

In this lab you will learn how to insert simple error handling into the JMart script and to throw custom error messages as well as catching basic JavaScript messages.

To complete this lab, you'll need to work through three exercises, and optionally, the third exercise for the Venkman Debugger:

- Bullet-Proof Functions
- Nested Try/Catch
- Venkman Test

Each exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

# Bullet-Proof Functions

## Objective

In this exercise, you will set up error handling in the two main functions of the JMart application. Most functionality will be encapsulated, and each try/catch block will utilize a finally block as well.

## Things to Consider

- While this exercise captures almost any functionality, it is not necessary to wrap an entire function into try/catch blocks when the code obviously will not produce errors, such as in default variable declarations.
- You can throw any object as an error. It is in better style to actually throw an error of type Error() object: var customerror = new Error('error message');
- To help ensure cross-browser compatibility, stick to using the basic name and message properties of an error object.

## Step-by-Step Instructions

1. Open **JMartSale.js** in this lab's directory. **JMartSale.html** contains a form and a **Click for new products!** button, which launches the main script. You will not have to edit this file; you will simply use it to test the application.
2. The first task is to modify the code in such a way as to break the application. Under the categories variable declaration near the top of **JMartSale.js**, add another line to break it: **categories = “”**. This will not throw a JavaScript error, but will produce undesirable results in the alert when the script executes. This is because the application assumes that the categories variable references a valid string object, and encounters problems when it tries to perform methods on the empty string that categories references.

```
<!--  
//J-Mart Stock Display  
var categories = "Batteries,Soap,Magazines,Tires";  
categories = "";
```

3. To bullet-proof the functions, it is essential to determine where they are most likely to break. In the **createArray()** function, the weak spots are the incoming parameters. In **showArray()**, the weak spot is the call to **createArray()**, because a broken or null array could be returned. Surround the entire contents of the **createArray()** function in a try block, followed by a catch block and a finally block.

```
function createArray(catlist, products, delim) {  
    try {  
        // surround the current contents of the  
        // createArray() function in the try block  
    } catch() {  
    } finally {  
    }  
}
```

4. Move the line **return aresult;** into the finally block. The statement in the finally block is guaranteed to execute, regardless of whether an error occurs or not.

```
function createArray(catlist, products, delim) {  
    try {  
        // contents of original createArray() function  
    } catch() {  
    } finally {  
        return aresult;  
    }  
}
```

5. For the catch block, specify **err** as the error parameter that is passed to the catch method, **catch (err)**. Within the catch block, add an alert: **createArray (parent error handler) threw “ + err.name + “:\n” +**

**err.message.** On the next line, set **aresult** to null so that it returns a null value: **aresult = null.**

```
function createArray(catlist, products, delim) {
    try {
        // contents of original createArray() function
    } catch(err) {
        alert("createArray (parent error handler) threw "
            + err.name + ":\n" + err.message);
        aresult = null;
    } finally {
        return aresult;
    }
}
```

6. In the try block in createArray(), add an if block: **if (catlist == "") {}** under the third line of the function (**var cats = catlist.split(delim);**). On the first line in the if block, add a statement to define a new Error object in the event that catlist is an empty string: **var noCatError = new Error("No categories were available.");**. On the following line, add the statement **noCatError.name = "Category Definition Error";** to give the new error object a specific name. On the last line of the if block, add the statement **throw noCatError.** This ensures that the error is caught if an empty string category is defined, much like the one you created in Step 2.

```
try {
    var cats = catlist.split(delim);
    if(catlist == "") {
        var noCatError = new Error("No categories were
            available.");
        noCatError.name = "Category Definition
            Error";
        throw noCatError;
    }

    var aresult = new Array();
```

7. Surround the contents of the showArray() function in a try/catch/finally block, placing the code in the try section, to catch any errors that may arise.

**Lab 9:**  
**Error Handling**

---

```
function showArray() {  
    try {  
        // surround the current contents of the  
        // showArray() function in the try block  
    } catch (err) {  
    } finally {  
    }  
}
```

8. Move the last line, **alert(msg)**, to the finally block to ensure that it always executes. For the catch block, catch the variable **err** with the statement **catch (err)** and a single statement in the catch block to customize the error message that will be generated in the event of an error: **msg = "showArray threw " + err.name + ":\\n" + err.message;**

```
function showArray() {  
    try {  
        //CURRENT CONTENTS OF showArray()  
    } catch (err) {  
        msg = "showArray threw " + err.name + ":\\n" +  
              err.message;  
    } finally {  
        alert(msg);  
    }  
}
```

9. After the **marray** is defined at the top of showArray(), add an if block to check whether marray exists: **if (!marray) {}**. For the first statement in the if block, add a statement to create a new error in the event that marray does not exist: **var nullarray = new Error("Array evaluated to null.")**.

```
try {  
    var marray = createArray(categories,  
                            aProducts, ",");  
    if (!marray) {  
        var nullarray = new Error("Array evaluated  
                                to null.");  
    }  
} catch (err) {  
}
```

10. On the next line of the if block, add a statement to set a specific name for the new Error object: **nullarray.name = “Null Array Error”**. Finally, on the last line of the if block, add a statement to throw the error: **throw nullarray**.

```
if (!marray) {  
    var nullarray = new Error("Array evaluated  
    to null.");  
    nullarray.name = "Null Array Error";  
    throw nullarray;  
}
```

11. Test the application by launching **JMartSale.html**. Upon pressing the **Click for new products!** button, an alert indicating a Category Definition Error should appear, followed by a Null Array Error alert as in Figure 11.

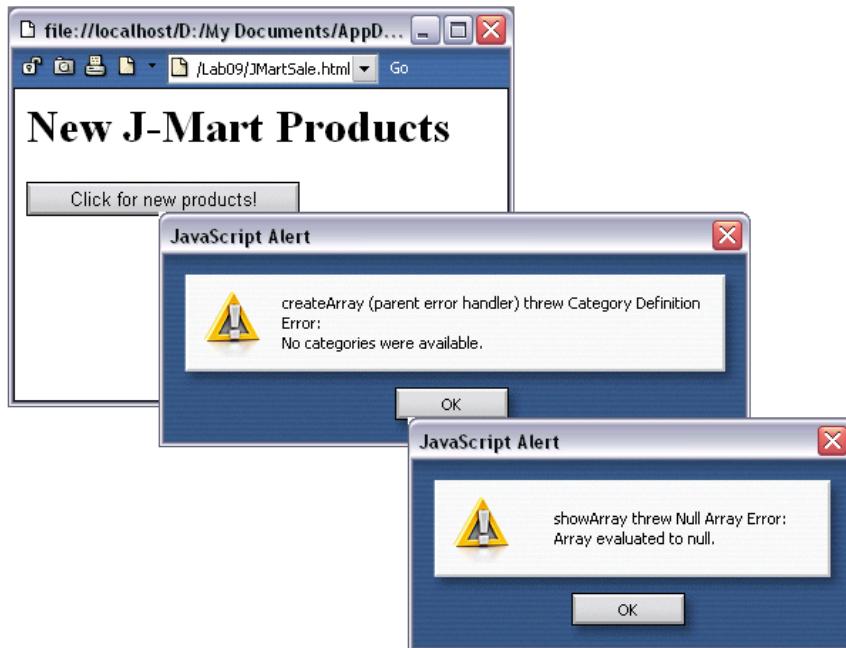


Figure 11. Errors caught in exception handling for JMartSale.js.

# Nested Try/Catch

## Objective

In this exercise, you will capture errors on a lower level within the `createArray()` function using a nested try/catch block.

## Things to Consider

- If an error is thrown from a nested try/catch block, via the event model, it bubbles up to the top-level try/catch block.

## Step-by-Step Instructions

1. Open the **JMartSale.js** file you completed in the first exercise. The first thing to do is comment out the line from the first exercise that produces the error. At the top of the file, right under the `categories` declaration, `categories` was set to an empty string. Comment out that line:

**NOTE** The completed files are set up for the first exercise. You must comment the line that sets the `categories` to an empty string as shown below, and perform Step 2 on the completed file before you run it.

```
<!--  
//J-Mart Stock Display  
var categories = "Batteries,Soap,Magazines,Tires";  
//categories = "";
```

2. In the `createArray()` function, break the for loop by adding `+ 10` after the `cats.length` in the loop's expression: `for (ca = 0; ca < cats.length + 10; ca++)`. When the code tries to access an array position beyond the length of the array itself, it will generate an error.

```
ArrayLoop:
for (ca = 0; ca < cats.length + 10; ca++) {
}
```

3. In the for loop located in `createArray()`, add a try/catch block under the line: `tmparray[0] = cats[ca]` near the top of the function:

```
ArrayLoop:
for (ca = 0; ca < cats.length; ca++) {
    var tmparray = new Array(2);
    tmparray[0] = cats[ca];
    try {
    } catch () {
    }
    aresult[ca] = tmparray;
}
```

4. Move the two lines that define the `tmparray` variable into the try part of the block (`tmparray[1] = products[ca].split(delim);` and `tmparray[1] = tmparray[1].sort();`). Catch the variable `err` in the catch block using the statement `catch (err)`, and in the body of the catch block, add the line `throw err;`

```
try {
    tmparray[1] = products[ca].split(delim);
    tmparray[1] = tmparray[1].sort();
} catch (err) {
    throw err;
}
```

5. Now it is time to test the application. When the **Click for new products!** button is clicked, the nested try/catch that you just added will throw an error. Remember that there are actually only four entries in the categories array, and therefore the length of the array is 4. Since you add ten to the loop counter, the code will try to access an index position that exceeds the array size, and an error will be generated when the for loop code tries to access the fifth element in the categories array. The catch block in `createArray()` will catch the error and process it, thereby displaying the alert dialog box. Finally, `showArray()`, which calls the `createArray()`

**Lab 9:**  
**Error Handling**

---

function, will throw an error as well because of the null results that createArray() returns. The error results are shown in Figure 12.

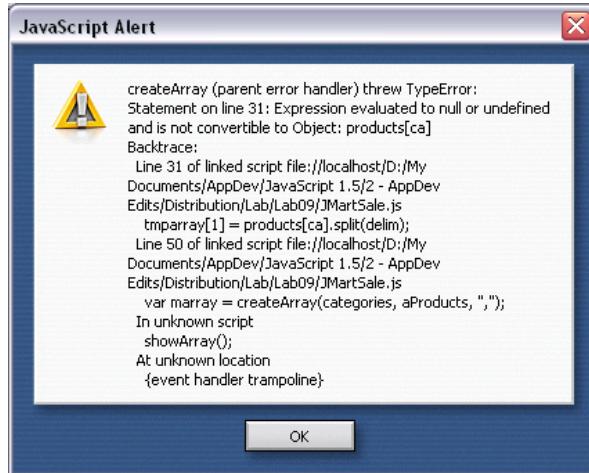


Figure 12. An error thrown by a nested try/catch and caught by a parent try/catch in createArray.

# Venkman Test

## Things to Consider

- Venkman can be launched with the url: “x-jsd:debugger”. This can also be used as a bookmark in Mozilla, Netscape, or Phoenix/MozillaFirebird.
- The Loaded Scripts panel allows you to browse all of the currently loaded scripts.
- Turning on Profile records all actions and variables after it has been activated, and you can then save them to a file from the Profile menu.
- To set a break point, click in the left “gutter” area of the Source code panel, on the line you want to stop on.
- To debug a script again, refresh it in the browser, and do any necessary actions to launch functions that contain break points.

## Step-by-Step Instructions

1. Open **JMartSale\_venkman.html** and **JMartSale\_venkman.js** in this lab’s directory.
2. Load JMartSale\_venkman.html in the browser in which you have installed Venkman (Mozilla, Netscape, Firebird, etc...).

**NOTE** Do not cheat and use the JavaScript console, because this is a debugger test!

3. You will notice that exceptions are thrown when the **Click for new products!** button is clicked. Open up Venkman, and search for the problem. Set break points within functions to figure out where the problem may be by investigating the breakpoints once the application has been run. The error to fix is shown in Figure 13.

**Lab 9:**  
**Error Handling**

---



Figure 13. Find the root cause of this error.

When the browser encounters the error, it will stop executing the code and open the Venkman debugger, as shown in Figure 14.

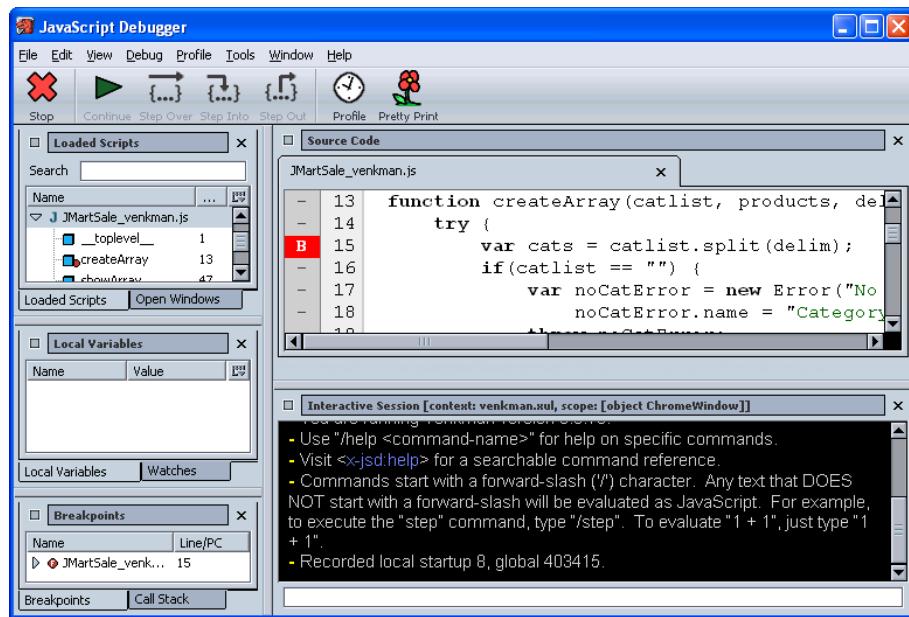


Figure 14. Debugging JMarsSale\_venkman in the Venkman debugger.

# Custom Objects

## Objectives

- Find out how Functions are used and their part in data type creation.
- Create methods for function objects.
- Learn how JavaScript objects were designed in the past.
- Understand function arguments and parameters.
- Learn to set up properties and prototype properties and methods.
- Learn the best practices for code organization.

# Functions

At this point, you have some exposure to JavaScript functions. Functions are the building blocks of organized JavaScript code and can be used to perform a wide range of useful operations. The strength of JavaScript functions is that they provide a basic block of functionality that can be called from any Web page that has access to them. They can accept parameters, use them in complex operations, and return results to the caller. Functions can even work with variable references that have been passed in to manipulate properties of the original object being referenced.

In this chapter, you will see that functions can serve a different role: as objects that you can define. Functions have a very important keyword, called *this*, that provides scripters with a reference to the function itself. You will see that it is possible to define custom properties for your functions by using the *this* keyword. Another object-enabling aspect of functions is that you can assign a function to a variable. This essentially allows you to create instance objects of that function, each of which has its own set of properties and can be referenced independently by its variable name.

The ability to create discrete function objects can help you simplify what might otherwise be fairly complex tasks. For example, by defining properties in a function, you can create data structures to contain information about the real-world objects you are trying to model in code. You can then use these function objects in arrays, or assign one function object to a property of another.

## Functions as Objects

To introduce the concept of functions as objects, consider the following simple function:

```
function animal(numLegs, hasTail) {  
    this.legs = numLegs; // Leg Property  
    this.tail = hasTail; // Tail Property  
}  
  
var cat = new animal(4, true);  
alert("Cat:\nLegs: " + cat.legs + "\nTail: " + cat.tail);
```

You can see here how simple it is to create a JavaScript object. A function object can contain custom properties and have custom methods to manipulate the object's data.

## Functions on the Fly

JavaScript also allows you to define functions on the fly. Sometimes, due to user input or other factors, you may want to have your script create functions in-line with the code block in which they are called. This is a useful technique if you want to quickly define a function that you would never need to reuse elsewhere in your code. With this technique, you simply assign a new function to a variable, as in the following code snippet:

```
var rand = new Function("num",
    "return Math.floor(Math.random() * num)");
```

Notice that the last parameter must be a string containing the statement that the function will execute. Any parameters defined before that string are treated as the actual parameters that will be passed in to the function. This function is scripted to accept only one parameter, called num, but any number of parameters may be defined. If you use this method, make sure that all the parameters are surrounded by quotes.

## Nested Functions

Functions can also be nested within other functions. If you know that you have a block of function code that will only be called within the context of another function, then you can nest them to encapsulate the functionality where it is needed. In the following example, the rand() function is nested within the displayNums() function.

```
function displayNums() {
    var c = 0;
    for (i = 0; i < 1280; i++) {
        c += rand(i); // rand() nested w/in displayNums()
    }

    function rand(num) {
        return Math.floor(Math.random() * num);
    }
}
```

However, you may find that the rand() function in the previous example needs to be called from a variety of places in your script. If this is the case, then you would want rand() to be a top-level function in your script.

**NOTE** A function can also be called from within another function. This setup allows for custom methods to be assigned within the boundary of the function where the Function's this property is applicable. When assigning a function to a property, you set it up much like a prototype, without the parentheses. A nested function can also serve as a property of the function, as an internal method. This topic is covered later in the chapter.

# Variables and Arguments

You can also pass parameters to JavaScript functions. Within the parentheses that appear after the function name in your function definition, you can add a comma-separated list of parameter names. Variables that are passed in to a function are references to the actual objects that the variables represent. This means that if a parameter called document.form1.textfield is passed in to the function that changes this parameter's value, you will change the form object's properties when you manipulate txtfield, as shown in the following code snippet.

```
<form name="frm1">
    <input type="text" name="textfield" value="Ralph"/>
    <input type="button" value="Change Name to Bob"
        // pass txtfield to changeName() as a parameter
        onclick="changeName(document.frm1.textfield, 'Bob')"/>
</form>

<script language="JavaScript">
    function changeName(nametextField, newname) {
        // nametextField variable references txtfield object
        nametextField.value = newname;
    }
</script>
```

In the preceding example, the changeName() function changes the value of txtfield from Ralph to Bob and displays it visually on the form.

## Undefined Parameters

Functions offer scripters some flexibility in the way that they handle passed-in parameters. If you pass in a parameter that is not listed between the parentheses in that function's definition, the function will not throw an error. In fact, you can pass in as many undefined parameters as you want. In the background, all of the passed-in parameters are placed into an array called arguments, which is a property of the function. This comes in handy when you have multiple objects that you need to pass in and iterate through, as illustrated in the following code snippet:

```
function addthem() {
    var sum;
    for (a = 0; a < arguments.length; a++) {
        sum += arguments[a];
    }
    return a;
}

alert(addthem(1, 1, 1, 1));
```

The arguments array will also contain any defined parameters of the function, as shown in the following code block:

```
function testParam(hello, gbye) {
    for (a = 0; a < arguments.length; a++) {
        alert(arguments[a]);
    }
}
alert(testParam("test hello", "test goodbye"));
```

In the preceding example, the alert box will display the passed-in values. The arguments array will include a function's defined parameters as well as any undefined parameters that the scripter passes in. If, for example, the function defines two parameters and the caller passes in five parameters, the function will consider the first two parameters to be its defined parameters. The function can access the additional parameters from the arguments array. If too few parameters are passed in, the function assigns the passed-in parameters to its defined parameters in the order in which they are received, and assigns null to any remaining defined parameters.

Each function also includes a property called length, which returns an integer value representing the number of parameters contained in the function's arguments array. If, for any reason, you need to know the number of parameters that were passed in to a function, you would call the function's length method.

# Objects and Properties

In JavaScript, functions can have properties, which you can define using most JavaScript variable types. Arrays, strings, integers, and other custom objects are all commonly used as properties. By defining properties for your function object, you can create complex data structures that describe the “real-world” objects that your code needs to model.

Consider the following cracker function, which is defined in conceptual terms as a sort of “virtual” cracker. For the purposes of our code, the cracker function is a blueprint for creating cracker objects, and it defines the properties of a cracker that you want to track. A cracker can be cheesy or not cheesy, salty or not salty, and can include any other attributes you require. You might build the blueprint for a cracker function as follows:

```
function cracker(name, desc, broken) {
    this.name = name || "unknown";
    this.description = desc || "unknown";
    this.broken = broken || false;
}
```

You establish properties of a function by using the function’s *this* keyword. You probably noticed that the parameter names that are defined for the cracker function are identical to the names of the cracker object’s properties. Prefacing the object’s properties with the keyword *this* helps the JavaScript interpreter distinguish between the properties and the corresponding parameter variables that you want to be able to assign to them via the function call.

**TIP:** In a JavaScript object, you can use the OR operator, `||`, to define default values for a property; these values will be used if no corresponding parameters for the property are passed in.

This sort of function is also known as a *constructor*, because it is used to construct instances of cracker objects. The assignment statements within the constructor allow you to define the specific properties of a given cracker object. When you want to create a new cracker object, you call the constructor function and pass in parameter values for each of the properties that you defined, as shown in the following code snippet:

```
var cheesycracker = new cracker("Cheezy",
    "A broken cheddar cracker", true);
alert(cheesycracker.name);
```

Note that the preceding example assigns the results of the function call to the variable `cheesycracker`. The `cheesycracker` variable refers to a fully functioning instance of a cracker object, with properties that can be manipulated in script, much like the properties of standard JavaScript objects. Each subsequent cracker object that you create will have its own variable name, with a distinct set of properties that you can access via the following syntax:

`[variable name].[property name]`

The alert box in the previous example displays the `name` property of the `cracker` object using this syntax:

```
alert(cheesycracker.name);
```

As an alternative to creating objects from a function, you can also define them using the `new Object()` syntax:

```
var airplane = new Object();
airplane.name = "Big Man";
airplane.type = "Boeing 747";
airplane.currentdest = "Toronto";
airplane.airline = "Air Canada";
```

In the preceding code snippet, the first statement defines `airplane` as a new `Object`, and the subsequent statements define the properties of the `airplane` object on an ad hoc basis. In this case, the `airplane` object will include properties that specify its name, type, current destination, and airline. This technique is helpful when you need only one instance of this object type, and you want to define it on the fly. However, if you need to create several objects of this type, it is typically more efficient to define the structure in a separate function. This enables you to define the function once, and then reuse it later to create additional instances of this object type.

Another method for creating objects is available to Internet Explorer 4+ and Netscape Navigator 4+ scripters:

```
var truck = {american_made:true, mileage:65000,  
make:"Ford", model:"F-150"};
```

This is known as literal syntax, and is also useful when you need to create single instances of simple objects on the fly. The truck object will contain properties that specify whether it is american\_made and its mileage, make, and model. As with the new Object() example, the literal syntax technique is most often used when you only need one instance of a truck, rather than multiple truck objects.

# Objects and Methods

Function objects can also include methods. A method is an element of a function object that calls another function. The function that the object's method calls can be defined either inside or outside the object.

For example, suppose you want each instance of cracker to return a string of all of its properties. You could create a function that builds the string, inserts the properties of the current object (`this.name`, etc.), and includes a return statement to send the result back to the caller.

*See **Function Methods.js***

To associate a function with an object's method, simply declare the new method within the object and assign the desired function to it, omitting the parentheses, as shown in the following code snippet:

```
function airplane(id, type, carrier, name, inflight) {  
    this.id = id || Math.floor(Math.random() * 5000);  
    this.type = type || "747";  
    this.carrier = carrier || "Delta";  
    this.name = name || "Super Jet";  
    this.inflight = inflight || false;  
    this.show = show;  
  
    function show() {  
        var tmp = "Information for airplane " + this.id;  
        tmp += "\nType: " + this.type;  
        tmp += "\nCarrier: " + this.carrier;  
        tmp += "\nName: " + this.name;  
        tmp += "\nIs in flight: " + this.inflight;  
        tmp += "\n";  
        return tmp;  
    }  
  
    var airplane1 = new airplane(0, "", "", "", true);  
    alert(airplane1.show());  
    //Result is the contents of 'tmp' in the show function.  
}
```

In this case, the `show()` method references a nested function, which is declared within the block of code in which the `airplane` function is defined. You could also create this function outside the `airplane` function and the reference would work equally well, as long as you define the function before the first `airplane` object is created. In this second scenario, you could reuse the `show()` function as a prototyped method on some other object, as long as it contains the same properties. The main consideration in deciding whether to use a nested function or an external function is whether that function's code could be reused with another object in the future.

## Arrays, Objects, or Object Arrays?

With the flexibility of JavaScript objects and arrays, it may be difficult to understand which to use when. An array is useful when you need a structure that allows you to store and iterate through large quantities of unrelated data. (It is more prudent to use objects when you need to work with a series of data elements that, when aggregated, comprise a more complex entity.) Finally, it is good practice to use arrays to store and iterate through complex objects of the same type.

See [Hangar.html](#)

Consider the following hangar example:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html public
  "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
  <head>
    <title>Lab 10 - Animal Crackers</title>
  <style>
    .divstyle {
      background-color: #EFEFEF;
      border: 1px solid #000000;
      overflow: auto;
      height: 250px;
      width: 250px;
    }
  </style>
  </head>
```

```
<body bgcolor="#FFFFFF">
<script language="JavaScript">
<!--
var hangar = new Array();

function airplane(id, type, carrier, name, inflight) {
    this.id = id || Math.floor(Math.random() * 5000);
    this.type = type || "747";
    this.carrier = carrier || "Delta";
    this.name = name || "Super Jet";
    this.inflight = inflight || false;
    this.show = show;

    function show() {
        var tmp = "<b>Information for airplane " +
                  this.id;
        tmp += "</b><br/>Type: " + this.type;
        tmp += "<br/>Carrier: " + this.carrier;
        tmp += "<br/>Name: " + this.name;
        tmp += "<br/>Is in flight: " + this.inflight;
        tmp += "<br/>";
        return tmp;
    }
}

//Create a fleet of 50 airplanes
function createFleet() {
    hangar = new Array();
    for (i = 0; i < 50; i++) {
        hangar[i] = new airplane(i + 1, "7" + i,
                               "Dave Airlines",
                               "Airplane #" + (i + 1),
                               (Math.floor(Math.random() * 2)) ? true :
                               false);
    }
}

function showFleet() {
    var html = "";

```

```
for (h = 0; h < hangar.length; h++) {
    html += hangar[h].show();
    html += "<br/><br/>";
}
//This is an object detection scheme, allows
//browser detection based on specific object
//availability.
if (document.all) {
    document.all.hangardiv.innerHTML = html;
} else {
    //Must be Netscape or Mozilla
    var tmp = document.getElementById("hangardiv");

    tmp.innerHTML = html;
}
}
//-->
</script>
<form>
    <input type="button" value="create fleet"
           onclick="createFleet()"/>
    <input type="button" value="show fleet"
           onclick="showFleet()"/>
</form>
<div id="hangardiv" class="divstyle"></div>
</body>
</html>
```

Figure 1 shows the results of the Hangar example rendered by the Opera 7 browser.

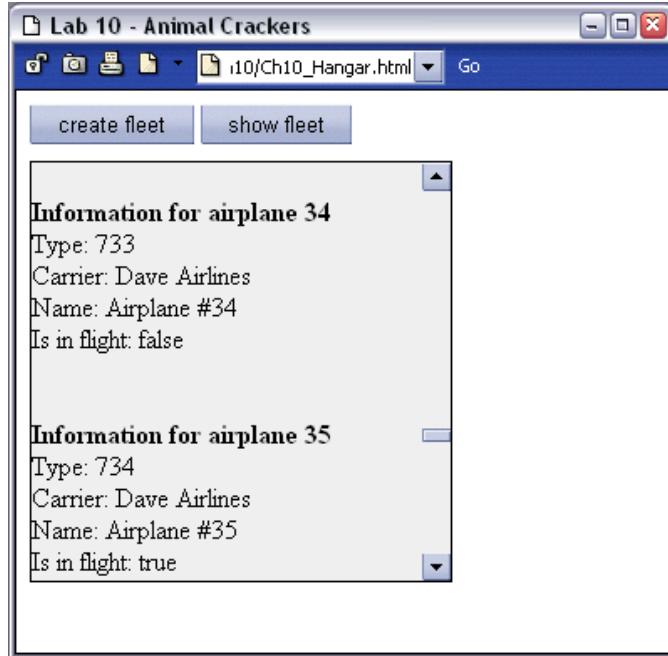


Figure 1. The airline hangar example in Opera.

**TIP:** The preceding example uses a css (.divstyle) class applied to a div tag to achieve the iframe-like appearance. This is a handy display technique in newer browsers.

As you may have already deduced, it is a better practice to create internal object methods to manipulate that object's properties rather than trying to manipulate them via an array's prototyped methods. This places the onus for maintaining the object's data integrity on the object itself rather than on an external function, and distinguishes an array as a container for storing objects.

## Prototypes

A function's prototype method is identical in syntax and function to an array prototype and allows you to create custom methods and values for the function. For instance, if you need to provide a mechanism to show the value of a certain property of a function, create a separate function and prototype it to a named property of the main function.

```
airplane.prototype.alertname = alertit;  
  
function alertit() {  
    alert(this.name);  
}  
  
hangar[3].alertname();
```

Consequently, you can create internal methods for a function object by prototyping rather than using the direct assignment technique described earlier in this section.

More advanced functionality for prototyping is available, however, the browsers are not at all standardized on prototyping. This creates problems with cross-browser scripting that you can avoid by creating well-formed JavaScript objects.

# Method Overriding

In JavaScript, an instance of a function object can override methods defined by the parent function. This simply means that the method is redefined to generate different results.

*See Method  
Overriding.js*

Consider the airplane function and the object instances, airplane1 and airplane2, in the following example:

```
function airplane(id, type, carrier, name, inflight) {  
    this.id = id || Math.floor(Math.random() * 5000);  
    this.type = type || "747";  
    this.show = show;  
  
    function show() {  
        var tmp = "Information for airplane " + this.id;  
        tmp += "<br/>Type: " + this.type;  
        tmp += "<br/>";  
        return tmp;  
    }  
}  
var airplane1 = new airplane();  
var airplane2 = new airplane();  
airplane1.show = newShow;  
  
function newShow() {  
    return this.type;  
}  
document.write(airplane1.show());  
//Result is just the type, 747  
document.write(airplane2.show());  
//Result is the original parent's show method.
```

You can override the function for each instance of the parent function. If you assign an object to another object variable of the same type, whatever properties or methods you change in either object will be reflected in the other object.

# Best Practices

Before venturing any further into function objects, you will find it extremely helpful to understand some best practices that will help you to better organize your code. These practices include code refactoring, the template technique, and using libraries to organize your code for optimal reuse.

## Code Refactoring

Code refactoring is a modern approach to code organization, and merits consideration as a way to help you develop sound coding practices. The idea is to eliminate complexity in your code by organizing discrete segments of related code into descriptively named functions. The greatest advantage of refactoring is that your code will be easier to maintain and change later because of its modular structure.

One of the best ways to take advantage of a function is to define one that handles repetitive code. For example, if your script must repeatedly convert a number to a string and apply a currency format to it, you would not want to type out the code to perform this conversion multiple times in your script! It would be far simpler to create a function that accepts the number and possibly some input regarding the currency format to apply, and returns the formatted string.

*See [Unrefactored Code.js](#)*

The following code snippet illustrates code that is implemented in a repetitive procedural fashion, which makes it difficult to isolate statements that go together:

```
//Un-refactored code:  
for (c = 0; c < 3; c++) {  
    //Set cracker & cracker image properties  
    var newcracker = new cracker();  
    var crackerpic = new crackerimg();  
    //Setup properties for cracker  
    crkr.id = loopid;  
    crkr.name = gNames[rand(gNames.length)];  
    crkr.description = gDesc[rand(gDesc.length)];  
    crkr.broken = (rand(2) == 1) ? true : false;  
    //Setup properties for images  
    var apic = gImgs[rand(gImgs.length)].split("|");  
    crackerpic.src = apic[0];  
    //If broken, add 'b_' prefix to source.  
    if (newcracker.broken) {  
        crackerpic.src = "b_" + apic[0];  
    }  
    crackerpic.author = apic[1];  
    crackerpic.type = apic[2];  
  
    //Add cracker image to cracker  
    newcracker.img = crackerpic;  
  
    //Stuff cracker into crackers array  
    gTresCrackers.push(newcracker);  
}
```

See  
**RefactoredCode.js**

As you can see, it requires a fair amount of effort to trace your way through the preceding script, to see exactly what it is doing. With a little refactoring to separate related chunks of code into cohesive functions called setCrackerProperties() and setImageProperties(), the following version is much easier to comprehend:

```

for (c = 0; c < 3; c++) {
    //Set cracker & cracker image properties
    var newcracker = new cracker();
    var crackerpic = new crackerimg();

    setCrackerProperties(newcracker, c);
    setImageProperties(crackerpic, newcracker);

    //Add cracker image to cracker
    newcracker.img = crackerpic;

    //Stuff cracker into crackers array
    gTresCrackers.push(newcracker);
}

//Utility functions for initCrackers
function setCrackerProperties(crrkr, loopid) {
    crrkr.id = loopid;
    crrkr.name = gNames[rand(gNames.length)];
    crrkr.description = gDesc[rand(gDesc.length)];
    crrkr.broken = (rand(2) == 1) ? true : false;
}

function setImageProperties(crkrimg, crrkr) {
    var apic = gImgs[rand(gImgs.length)].split("|");
    crkrimg.src = apic[0];
    //If broken, add 'b_' prefix to source.
    if (crrkr.broken) {
        crkrimg.src = "b_" + apic[0];
    }
    crkrimg.author = apic[1];
    crkrimg.type = apic[2];
}

```

Note that refactoring was not essential in the preceding example, due to the brevity of the code. However, it should help you to see how your code can be much more readable in its reorganized form. In addition, if an image or cracker needs new properties and methods, it is easier to find all of the places where you will need to add or change code.

## Template Technique

Although you can go back through your existing code and refactor it, you may find it easier to use code organization techniques in your up-front design. While it can also be used after the fact, the Template technique is particularly useful in the design phase, when you are considering what your program needs to do.

Applying the Template technique is very similar to creating an outline before you write a document. When you design a complicated function, start by considering the discrete steps that it must perform. Think of each step as a separate sub-function, and begin writing each one as a “stub,” without any code in the braces. At the start, simply give each sub-function a descriptive name that briefly expresses what it does; you will come back and fill in the details later.

The Template itself is nothing more than a parent function that calls the sub-functions for each step. When you write the parent function, simply have it call the sub-functions in the appropriate order. The descriptive sub-functions’ names will make it easy to see exactly what the parent function is doing, and can reduce or even eliminate the need for extensive comments in your code.

The following code snippet illustrates the general concepts of the Template technique. When buildCalendar() is called, it sets up the html string by calling numerous functions that perform individual tasks.

```
function buildCalendar(date) {  
    html += buildHeaderInformation();  
    getStartDay();  
    getEndDay();  
    // etc, etc  
    return html;  
}
```

```
function buildHeaderInformation() {    }

function getStartDay() {
    var startday = dayOfWeek(getMonth(), 1, year);
    return startday;
}

function getEndDay() {
    var endday = dayOfWeek(getMonth(), mnthLen, year);
    return endday;
}

...
```

Applying the refactoring and Template techniques may seem difficult at first, especially with a complex project. However, you'll find that the time you invest to become proficient at it is more than offset by the time you'll save designing, debugging, and maintaining your code in the future.

## Creating Organized Libraries

By this point, you probably have created external .js files, black-box functions, and a few abstract objects (airplane, cracker). The benefit of creating truly universal functions, and simple objects, is that you can create libraries from them to reuse in the future.

For example, over time you may have developed a number of useful functions that all work with date and time. Some may be simplified stopwatch functions or string converters for date formats, while others could be RegExp-based functions that evaluate dates, or even more obscure but highly useful functions that you'd hate to have to rewrite.

Well, there is no time like the present to get your code organized into libraries, so it will be easier to find these functions when you need them. To create a library, you should gather up all those related functions and place them in a custom .js file with a name that represents its contents, such as myDateTimeFunctions. You may even want to put this file into a central folder, with a directory name like myMethods, or something to that effect.

The next time you work on a page that could use one or more of these methods, you can simply copy and paste from the file to your Web site, tweak the methods as needed to implement any modifications, and call them from other JavaScript resources. This will enable you to take advantage of your existing scripts and you won't need to reinvent the wheel when similar scenarios arise.

# Summary

- Functions are the building blocks of organized JavaScript code.
- Functions can refer to their own properties and methods by using the *this* keyword.
- Data can be organized by using functions intelligently.
- New instances of a function are created by calling: var *instance* = new *functionName*();.
- Functions can be nested for functionality specific to the main function.
- Methods of custom functions are used by assigning a function to the methodname: *this.methodname* = *functionName*;
- Parameters of a function object can be used to define a new instance of the function object upon construction.
- Function objects can be stored in arrays for easily accessible collections of that data type.
- Methods and properties of an instantiation of a function object can be overridden to provide different results.
- Code refactoring is the process of condensing blocks of code intelligently into structured, descriptively named methods.

# Questions

1. Identify this syntax: var car = new Function {name:"unknown"};.
2. What is the process called of breaking long blocks of statements into separate functions that are well-named?
3. What part of a JavaScript object is actually a separate function that defines a common functionality?
4. How does the syntax look that allows you to call JavaScript from an XHTML link?
5. True/False: A child object overrides, or changes, the method of its parent. This, in turn, changes the parent's original method.
6. Why is prototyping not yet the best way to define methods?
7. What is the invisible array property of a function called?
8. Other than a function, you can define a JavaScript object on the fly by calling new \_\_\_\_\_.
9. If a method of an object provides functionality that is only useful to that object, what kind of function should you use?
10. You have created an object called cosmicObject. How do you create an instance of it?

# Answers

1. Identify this syntax: var car = new Function {name:"unknown"};  
**Literal Syntax**
2. What is the process called of breaking long blocks of statements into separate functions that are well-named?  
**Refactoring**
3. What part of a JavaScript object is actually a separate function that defines a common functionality?  
**A Method**
4. How does the syntax look that allows you to call JavaScript from an XHTML link?  
**<a href="JavaScript: functionName()">Link Text</a>**
5. True/False: A child object overrides, or changes the method of its parent. This, in turn, changes the parent's original method.  
**False**
6. Why is prototyping not yet the best way to define methods?  
**Browser Incompatibilities**
7. What is the invisible array property of a function called?  
**The argument array**
8. Other then a function, you can define a JavaScript object on the fly by calling new \_\_\_\_\_.  
**Object()**
9. If a method of an object provides functionality that is only useful to that object, what kind of function is a good idea?  
**A Nested Function**
10. You have created an object called cosmicObject. How do you create an instance of it?  
**var newInstance = new cosmicObject();**

# Lab 10: Custom Objects

**TIP:** Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You will find all the code in **AnimalCrackers.html** and **AnimalCrackers.js** in the same directory as the sample project. Also, the images you'll need are: **butterfly.gif**, **b\_butterfly.gif**, **elephant.gif**, **b\_elephant.gif**, **puppy.gif**, **b\_puppy.gif**, and **CrackerBox.gif**. While the lab can be constructed without the images, it is better to use them to see how the data translates to a visual interface. To avoid typing the code, you can cut/paste it from the source files instead.

# Lab 10 Overview

In this lab you will learn how organizing data into custom objects makes it easy to manipulate, display, and handle provided data.

To complete this lab, you will need to work through two exercises:

- Crackers, Photos, and their Methods
- Build a User Interface

Each exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

**NOTE** Each exercise has its own initCrackers function to launch the exercise from the <body onload='> event in AnimalCrackers.html. The first exercise launches with initCrackers() and the second exercise launches with initCrackersEx2(). To test the completed exercises, be sure to make the appropriate change to the onload attribute.

# Crackers, Photos, and their Methods

## Objective

In this exercise, you'll set up the objects for photos and crackers. In addition, you will create methods of these objects that will return useful information.

## Things to Consider

- Remember that default values can be assigned by using the **or** operator, which is represented by the double pipes: **this.src = src || "";**
- If a method is only going to be used by a particular object, it is appropriate to nest the function within the object's definition. Otherwise, define the function outside of the object definition to make it a global function, so that you can call it wherever it is needed.
- Broken crackers have a different image than complete crackers. They have the same base image name, but have the prefix **b\_** at the beginning of the filename.

## Step-by-Step Instructions

1. Open the **AnimalCrackers.js** file, or create a new one if you are feeling adventurous. The **AnimalCrackers.html** file sets up the interface for you, and looks for the code in the **AnimalCrackers.js** file. For this entire lab, it is important to place all the files within the same directory, including the images. Otherwise you will need to update any references to the image files within the XHTML and JavaScript code to reflect the correct path to the files.
2. Create a cracker object: **function cracker(id, name, desc, broken, crackimg) {}**, to set up the cracker and accept parameters for its initial values. Next, initialize the parameters using the **this.<parametername> = <passed in parameter name>** syntax to set the values for the cracker's id, name, description, its boolean indicator, broken, and an image file. Also, add two method definitions: **this.getType** and **this.show** with default values of "**unknown**" for both methods.

```

function cracker(id, name, desc, broken, crackimg) {
    this.id = id;
    this.name = name || "unknown";
    this.description = desc || "unknown";
    this.broken = broken || false;
    this.img = crackimg;
    this.getType = getType || "unknown";
    this.show = showCracker || "unknown";
}

```

3. Next, you must define the methods that were assigned to the **getType** and **showCracker** methods in Step 2. Create a nested function in the object called **getType()**, which will have only one statement: **return this.img.type;** that returns the type of image that the cracker has. The cracker image object itself will be defined in a later step.

```

function cracker(id, name, desc, broken, crackimg) {
    this.id = id;
    this.name = name || "unknown";
    this.description = desc || "unknown";
    this.broken = broken || false;
    this.img = crackimg;
    this.getType = getType || "unknown";
    this.show = showCracker || "unknown";

    function getType() {
        return this.img.type;
    }
}

```

4. Now, add **showCracker()** as a nested function. When **crackerobject.show()** is called, this is the function that executes. The first statement in the function should define the returned string: **var txt = “Animal Cracker” + this.id + “\n”;** The next line simply adds a dashed line using the statement **txt += “-----\n”;**, to provide a visual separation of the information. On the next line, add a statement to display the object’s name: **txt += “Name: ” + this.name + “\n”;** and its description: **txt += “Description: ” + this.description + “\n”;**, followed by statements to display the object’s type, **txt += “Type: ” +**

**this.getType()** + “\n”;, and broken boolean value, **txt += “Broken: ” + this.broken + “\n”;**. You will also want to append the information returned from the image object to the **txt** variable by adding **txt += “\n\n\n\n” + this.img.show();**. Again, the cracker image object will be defined in a later step. Finally, return the **txt** object using the statement **return txt;**.

```
function getType() {  
    return this.img.type;  
}  
  
function showCracker() {  
    var txt = "Animal Cracker " + this.id + "\n";  
    txt += "-----\n";  
    txt += "Name: " + this.name + "\n";  
    txt += "Description: " + this.description + "\n";  
    txt += "Type: " + this.getType() + "\n";  
    txt += "Broken: " + this.broken + "\n";  
    txt += "\n\n\n\n" + this.img.show();  
    return txt;  
}  
}
```

5. Create the **crackerimg** object outside of the cracker object using the statement **function crackerimg(src, desc, auth, type) {}**. As with the cracker object, add the following properties: **src**, **author** and **type**. In addition, add a reference to its **show** method with the statement **this.show = showImg;**

```
function crackerimg(src, auth, type) {  
    this.src = src || "";  
    this.author = auth || "unknown";  
    this.type = type || "unknown";  
    this.show = showImg;  
}
```

6. Create the crackerimg object’s internal **show** method with the statement **function showImg() {}**. Create the first string for the message **var txt = “Image Details:\n”;**, and add a visual separator line with the statement **txt += “-----\n”;**. Now, add statements to display specific

information about the image file, by appending cracker image **src**, **author** and **type** values to the **txt** string. Finally, return the **txt** string with the statement **return txt;**.

```
function crackerimg(src, auth, type) {  
    this.src = src || "";  
    this.author = auth || "unknown";  
    this.type = type || "unknown";  
    this.show = showImg;  
  
    function showImg() {  
        var txt = "Image Details:\n";  
        txt += "-----\n";  
        txt += "File: " + this.src + "\n";  
        txt += "Author: " + this.author + "\n";  
        txt += "Type: " + this.type + "\n";  
        return txt;  
    }  
}
```

7. Create a function to initialize some crackers, and call it **initCrackers()**. Create a new cracker image called **testcrackerimg**, using the statement **var testcrackerimg = new crackerimg("elephant.gif", "big elephant", "myself", "Elephant Cracker");**. Now create a new cracker called **testcracker**, using the statement **var testcracker = new cracker(123, "Cracker", "A tasty test", true, testcrackerimg);**. Finally, add a line to show the user information about the cracker: **alert(testcracker.show());**.

```
function initCrackers() {  
    var testcrackerimg = new crackerimg("elephant.gif",  
        "myself", "Elephant Cracker");  
    var testcracker = new cracker(123, "Cracker",  
        "A tasty test", true, testcrackerimg);  
  
    alert(testcracker.show());  
}
```

8. Open the **AnimalCrackers.html** file and add an onload event handler to the <body> tag that calls the **initCrackers()** method, and sets the page's background color to #FFFFFF.

```
<body onload="initCrackers()" bgcolor="#FFFFFF">
```

9. Open and test **AnimalCrackers.html**. When the page loads, you should get an alert box like the one in Figure 2, with the test crackers information. The page that comes up with the cracker box and the elephants will be completed in the next exercise.



Figure 2. The Opera 7 alert box with test cracker information.

# Build a User Interface

## Objective

In this exercise, you will finish the AnimalCrackers application. This will give you more practice working with defined custom objects.

## Things to Consider

- To swap an image to another image file, change its read/write **src** property.
- Remember to keep code refactored for easy maintenance.
- In this exercise, the cracker's id is obtained from a parameter of the img objects on the html page called **aindex**. This can be 0, 1, or 2 depending on the slot.
- The broken version of the image is the same filename, but with a “**b\_**” attached to the front: “elephant.gif” and “**b\_elephant.gif**”.

## Step-by-Step Instructions

1. Create a new init function for this exercise called **initCrackersEx2()**. In **AnimalCrackers.html** change the onload attribute in the <body> tag to “**initCrackersEx2()**”.

```
<body onload="initCrackersEx2()" bgcolor="#FFFFFF">
```

2. Return to your work from the previous example. Add a global variable (outside of all other function or object definitions) to the file with the statement **var gTresCrackers = new Array();** to hold three random crackers.

```
<!--
//Global Cracker Container
var gTresCrackers = new Array();
```

3. Within the new **initCrackersEx2** function, reset **gTresCrackers('gTresCrackers = new Array();')** to ensure that a new array is created. Following that statement, copy and paste the following arrays that provide lists of names, descriptions, and image information, which will be randomly selected when the exercise executes.

```
//Reset gTresCrackers
gTresCrackers = new Array();

//Options for random cracker generation.

var gNames = new Array("Frank", "Rudolph",
    "Poopiekins", "Yellow", "Akim", "Ralphy",
    "Poomba", "Royale with Cheese", "Rowdy Roddy",
    "Hulk");

var gDesc = new Array("Reliable Cookie.",
    "Fun to chomp on.", "allright...",
    "Crumbly goodness", "Sensational taste value!",
    "A bit soggy.", "Best cookie in all of the world",
    "Fabulous texture!", "Down-right stale!",
    "How old is this box?",
    "Allright, who poured milk in the box already?");

var gImgs = new Array(
    "elephant.gif|David Fitzhenry|Enormous Elephant",
    "butterfly.gif|Carolyn Lail Fitzhenry|Beautiful
    Butterfly", "puppy.gif|Noah Kriegel|Perky Puppy");
```

4. Under the new Arrays, create a new for loop that will go through three iterations: **for (c = 0; c < 3; c++) {}**. Within the body of the loop, define a new cracker using the statement **var newcracker = new cracker();** and a new image with the statement **var crackerpic = new crackerimg();**.

```
for (c = 0; c < 3; c++) {
    var newcracker = new cracker();
    var crackerpic = new crackerimg();

}
```

5. Set up two new utility functions, called **setCrackerProperties(crrkr, loopid) {}** and **setImageProperties(crkrimg, crck) {}**, that are outside of the loop, but within the **initCrackersEx2()** function. In the for loop, under the **newcracker** and **crackerpic** variables, add the following statements **setCrackerProperties(newcracker, c);** and **setImageProperties(crackerpic, newcracker);** to call the functions.

```

for (c = 0; c < 3; c++) {
    var newcracker = new cracker();
    var crackerpic = new crackerimg();

    setCrackerProperties(newcracker, c);
    setImageProperties(crackerpic, newcracker);
}

function setCrackerProperties(crrkr, loopid) {
}

function setImageProperties(crkrimg, crkr) {
}

```

6. Create a simple function, **rand(num) {}**, outside of the loop that accepts a numeric value parameter and returns a random number using the statement **return Math.floor(Math.random() \* num)** within the function.

```

function setImageProperties(crkrimg, crkr) {
}

function rand(num) {
    return Math.floor(Math.random() * num);
}

```

7. Go to the **setCrackerProperties** function, and add code to set the properties for the cracker using the statements **crkr.id = loopid;**, **crkr.name = gNames[rand(gNames.length)];**, **crkr.description = gDesc[rand(gDesc.length)];**, and **crkr.broken = (rand(2) == 1) ? true : false;**. This code will generate a cracker with properties selected randomly from the options that have been created in the **gNames** and **gDesc** arrays.

```
function setCrackerProperties(crrkr, loopid) {
    crrkr.id = loopid;
    crrkr.name = gNames[rand(gNames.length)];
    crrkr.description = gDesc[rand(gDesc.length)];
    crrkr.broken = (rand(2) == 1) ? true : false;
}
```

8. In the setImageProperties() function, create a new array called **apic** to hold the separate name, author, and description string values from a randomly selected gImg. Use the **split()** function to separate the name, author and description strings based on the “|” (pipe) character.

```
function setImageProperties(crkrimg, crkr) {
    var apic = gImgs[rand(gImgs.length)].split("|");
}
```

9. Set the **crkrimg** object’s source property, **src**, to the value of the string at the zero index of **apic**, which represents the filename. If the cracker is broken, append **b\_** to the beginning of the filename.

```
function setImageProperties(crkrimg, crkr) {
    var apic = gImgs[rand(gImgs.length)].split("|");

    crkrimg.src = apic[0];

    if (crkr.broken) {
        crkrimg.src = "b_" + apic[0];
    }

}
```

10. Finally, set the author value to index one of **apic**, and the type to index two of **apic**.

```

function setImageProperties(crkrimg, crkr) {
    var apic = gImgs[rand(gImgs.length)].split("|");

    crkrimg.src = apic[0];

    if (crkr.broken) {
        crkrimg.src = "b_" + apic[0];
    }

    crkrimg.author = apic[1];
    crkrimg.type = apic[2];
}

```

11. Return to the code in the for loop, and after the call to **setImageProperties()**, add a line to set the newcracker's image property, **img**, to crackerpic with the statement **newcracker.img = crackerpic;**. Then, on the next line, add a statement to push the completed newcracker object into the global cracker array: **gTresCrackers.push(newcracker)**.

```

for (c = 0; c < 3; c++) {
    var newcracker = new cracker();
    var crackerpic = new crackerimg();

    setCrackerProperties(newcracker, c);
    setImageProperties(crackerpic, newcracker);

    newcracker.img = crackerpic;
    gTresCrackers.push(newcracker);
}

```

12. To finish the user interface, you'll need to add a function to display the correct image on the Web page, and another function to generate a JavaScript alert message that contains the selected cracker image's properties. First create a new function, **packImageElements()** directly under the **rand(num)** function that you created in Step 6. Create an array of the image elements that are located on the **AnimalCrackers.html** page, with the statement **elements = new Array(document frm cracker0, document frm cracker1, document form cracker2);**.

```
function packImageElements() {  
    var elemnts = new Array(document.frm.cracker0,  
                           document.frm.cracker1, document.frm.cracker2);  
  
}
```

13. Each <img> tag in the XHTML has an aindex property associated with it, to represent its cracker's position in the global array, **gTresCrackers**. Create a for loop under the **elemnts** declaration that will perform three iterations, using the statement **for (f = 0; f < 3; f++)**. On the next line, set the **src** property of the current image in the array by adding the statement **elemnts[f].src = gTresCrackers[f].img.src**. Finally, set the image's aindex property to the current count with the statement **elemnts[f].aindex = f;**

```
function packImageElements() {  
    var elemnts = new Array(document.frm.cracker0,  
                           document.frm.cracker1, document.frm.cracker2);  
    for (f = 0; f < 3; f++) {  
        elemnts[f].src = gTresCrackers[f].img.src;  
        elemnts[f].aindex = f;  
    }  
}
```

14. Go back to the line after the for loop in the **initCrackersEx2()** function and call the **packImageElements()** function.

```
for (c = 0; c < 3; c++) {  
    var newcracker = new cracker();  
    var crackerpic = new crackerimg();  
  
    setCrackerProperties(newcracker, c);  
    setImageProperties(crackerpic, newcracker);  
  
    newcracker.img = crackerpic;  
    gTresCrackers.push(newcracker);  
}  
  
packImageElements();
```

Each image in **AnimalCrackers.html** has been given an onclick event that will fire a method called **selectedCracker(element)**. When this method executes, the image must pass in a reference to itself, which your code can use to get the information needed to display that cracker's properties.

15. Create the **selectedCracker(crckr)** function outside of all the other functions, preferably at the end of **AnimalCrackers.js**. The only line in this function will generate an alert that displays the selected cracker's information, using the statement  
**alert(gTresCrackers[crckr.aindex].show());**

```
function selectedCracker(crckr) {  
    alert(gTresCrackers[crckr.aindex].show());  
}
```

16. Test the application. Once it's running, you should be able to randomly set the crackers with the **Get More Animal Crackers!** button, and then click on each cracker to see its information. In addition, notice that if the selected cracker is broken, the program will display the broken cracker image, as in shown in Figure 3. If the selected cracker is not broken, the program will display the unbroken image, as shown in Figure 4.

**Lab 10:**  
**Custom Objects**

---

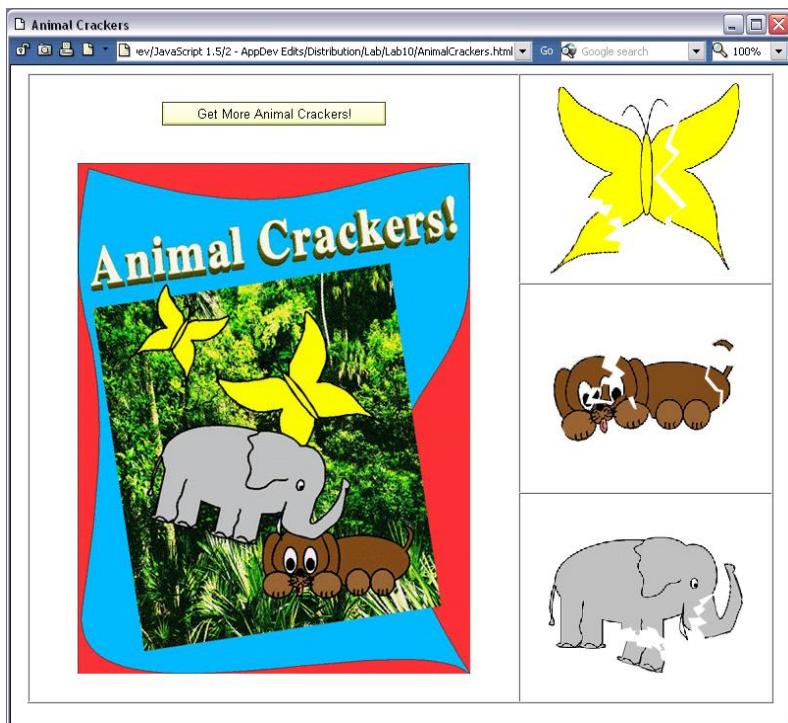


Figure 3. Random crackers in Opera 7, showing “broken” images.

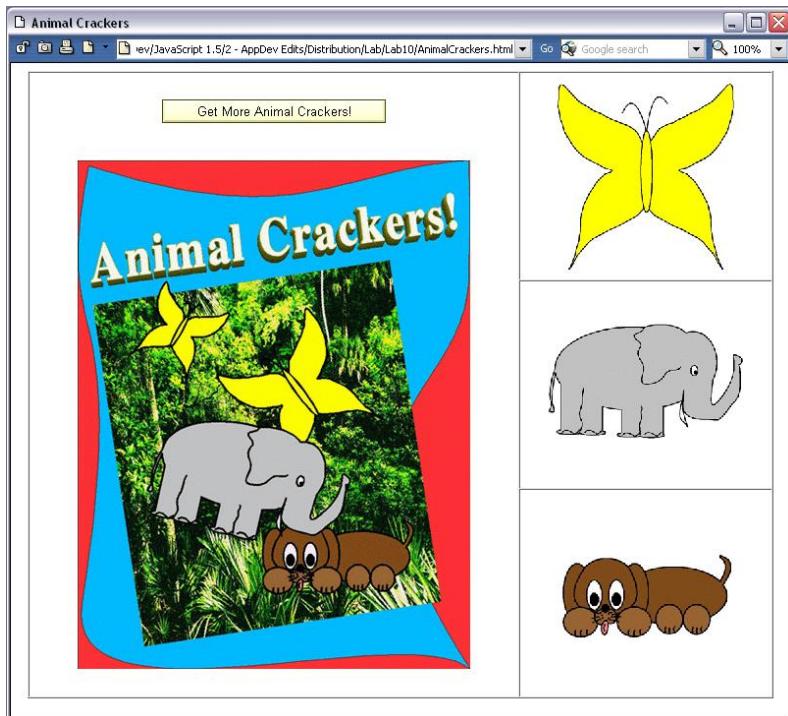


Figure 4. More random crackers in Opera 7, showing unbroken images.

# Appendix A: Resources

Feb 19 2008 3:29PM Dao Dung dungdq@edt.com.vn

For production evaluation only – not for distribution or commercial use.

**A-1**

Copyright © 2003 by Application Developers Training Company  
All rights reserved. Reproduction is strictly prohibited.



# Books

## ***JavaScript: The Definitive Guide***

**Author:** David Flanagan

**Publisher:** O'Reilly

**Comments:** An excellent reference manual. Great for looking things up, but not as strong to teach from.

## ***JavaScript Bible, 4<sup>th</sup> Edition***

**Author:** Danny Goodman

**Publisher:** Hungry Minds, Inc.

**Comments:** An all-around excellent book on JavaScript. Use as a reference or training supplement.

## ***Mastering Regular Expressions, Second Edition***

**Author:** Jeffrey, E.F. Friedl

**Publisher:** O'Reilly

**Comments:** Excellent Regular Expression reference.

## ***JavaScript: The Complete Reference***

**Author:** Fritz Schneider and Thomas A. Powell

**Publisher:** McGraw-Hill Osborne Media

**Comments:** Another excellent JavaScript reference manual.

# Web Sites

## <http://devedge.netscape.com/central/javascript/>

JavaScript Central: Official documentation and reference.

## <http://devedge.netscape.com/central/dom/>

Netscape's DOM Central Web site. Information about the DOM and standards-compliant DHTML.

## <http://www.w3.org/MarkUp/>

Official standards from W3C on HTML, XHTML, and general Web markup.

## <http://www.w3.org/DOM/>

Official DOM standards from the W3C.

## <http://www.ecma-international.org/>

Official site of the ECMAScript creators/maintainers.

## <http://www.mozilla.org/js/>

Mozilla.org is home of the Mozilla browser, Netscape's reference. This link is to their JavaScript section, which leads to information about JavaScript in Mozilla, and the C++ API for JavaScript.

## <http://devcentral.iftech.com/articles/Javascript/default.php>

Dev Central. The resource for many languages.

## <http://www.siteexperts.com/>

Excellent Q & A Web site on JavaScript, DHTML, and XHTML. Free membership and very active message board participants.

## <http://www.w3schools.com/>

Excellent site linked to W3C.org. Provides basic tutorials for various languages and technologies.

# Tools

## Free Editors

### jEdit

Link: <http://www.jedit.org>

The labs in this courseware were mostly constructed in jEdit 4.1 because it has superior code coloring for JavaScript (and many, many other languages). jEdit is an excellent tool to use because it is cross-platform (Java based), and provides an ever-expanding suite of plug-ins that make your coding experience much better. Favorite plug-in: Buffer Tabs. Creates a tabbed multi-document interface, which allows you to right-click on the tab and copy the path to the file to your Clipboard for pasting into a browser and testing (see Figure 1).

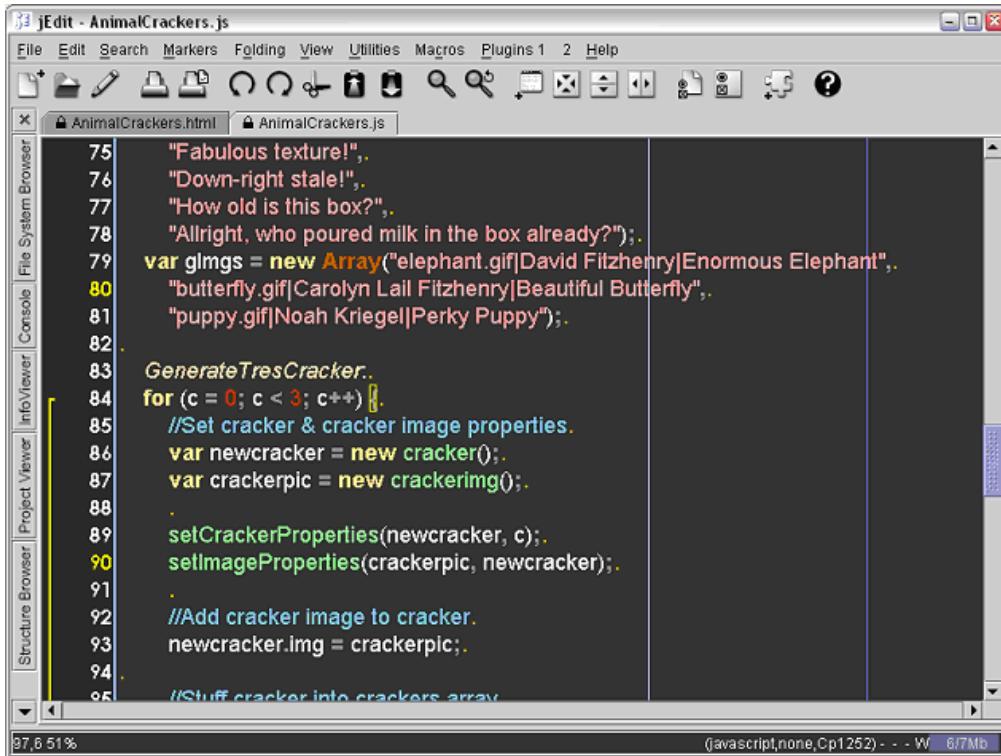


Figure 1. The jEdit 4.1 editor.

## Eclipse

**Link:** <http://www.eclipse.org>

Eclipse is IBM's modular Java editing environment. Eclipse is striking fear into its competition because of its capability as a Java IDE. It offers many advanced features like code refactoring and code insight, and you can't beat the price: it's open source and is available as a free download. While it is primarily used as a Java IDE, several JavaScript plug-ins are available that allow you to use it for coding in JavaScript. Figure 2 shows the Eclipse editor.

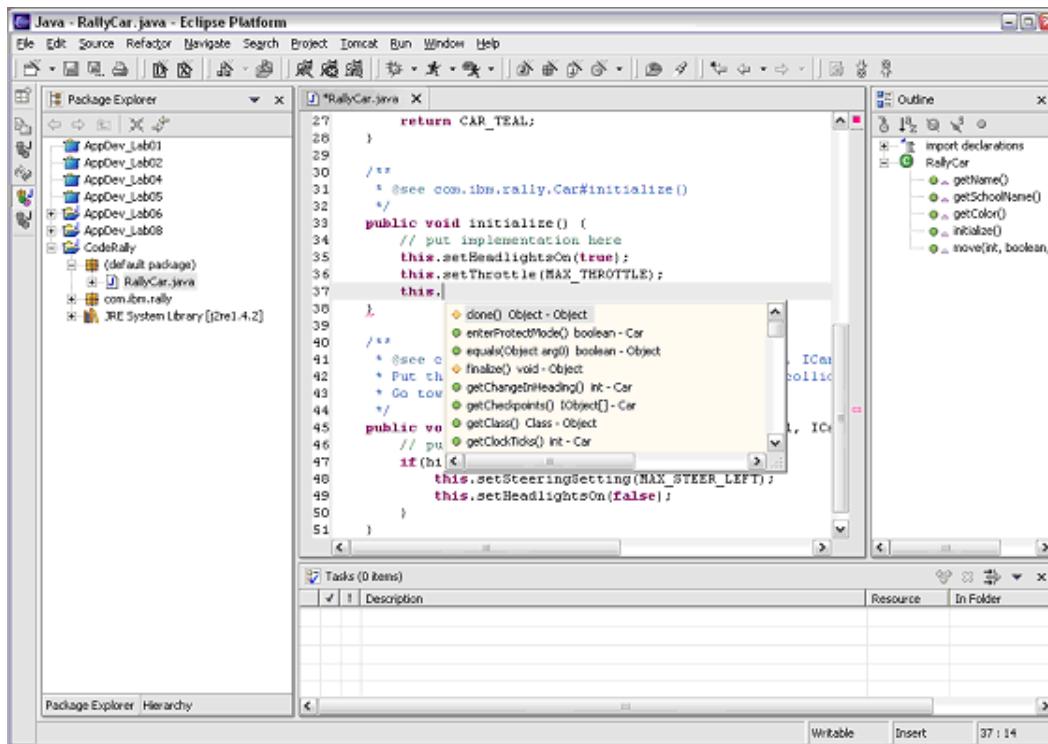


Figure 2. The Eclipse editor.

## Commercial Editors

### Macromedia Dreamweaver MX

**Link:** <http://www.macromedia.com/software/dreamweaver/>

Dreamweaver MX is possibly the best all-around Web site design and development tool. It offers comprehensive syntax highlighting for JavaScript editing, along with a good editable library of scripts. Dreamweaver has built-in DHTML support (behaviors), which adds tons of defined functionality to your

pages without writing script. Note that some of Macromedia's scripts are a little overblown, but you can customize them as needed (see Figure 3).

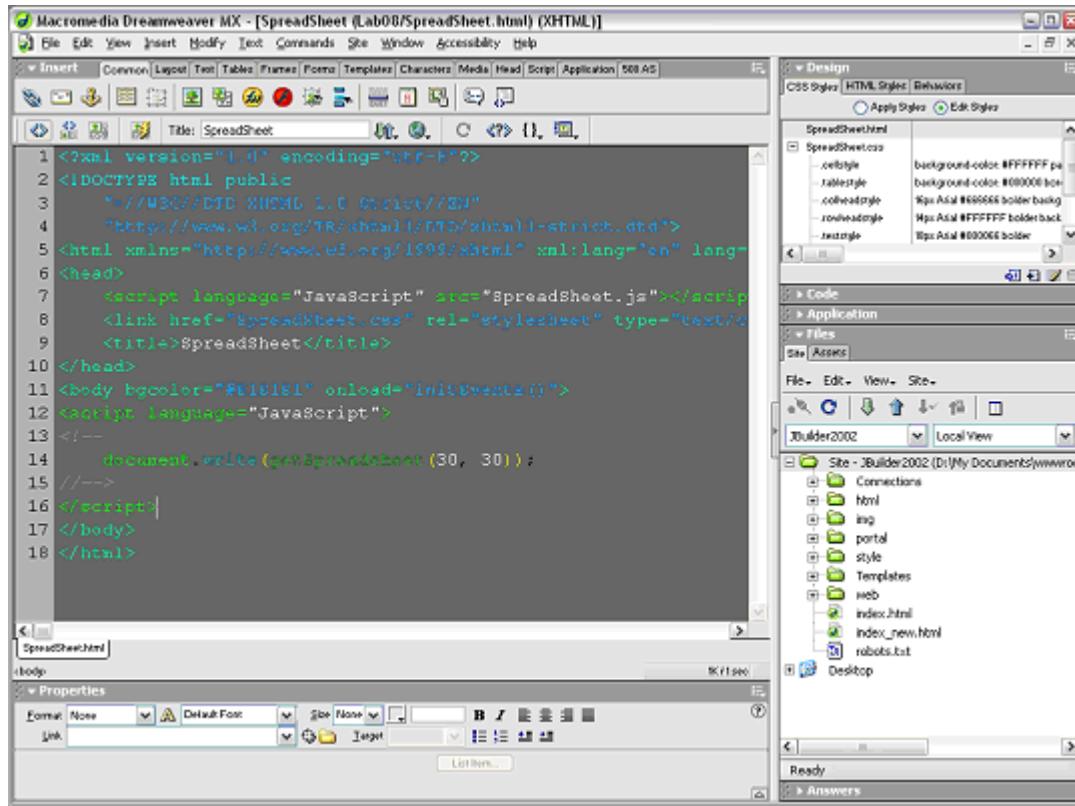


Figure 3. The Macromedia Dreamweaver MX editor.

## Antechinus JavaScript Editor

**Link:** <http://www.c-point.com/>

This editor offers many interesting options. It has code-completion for JavaScript, a built-in Web test environment and a browsable language spec. The language does seem to center more towards the Internet Explorer standard. Comes with many built-in scripts to try (see Figure 4).

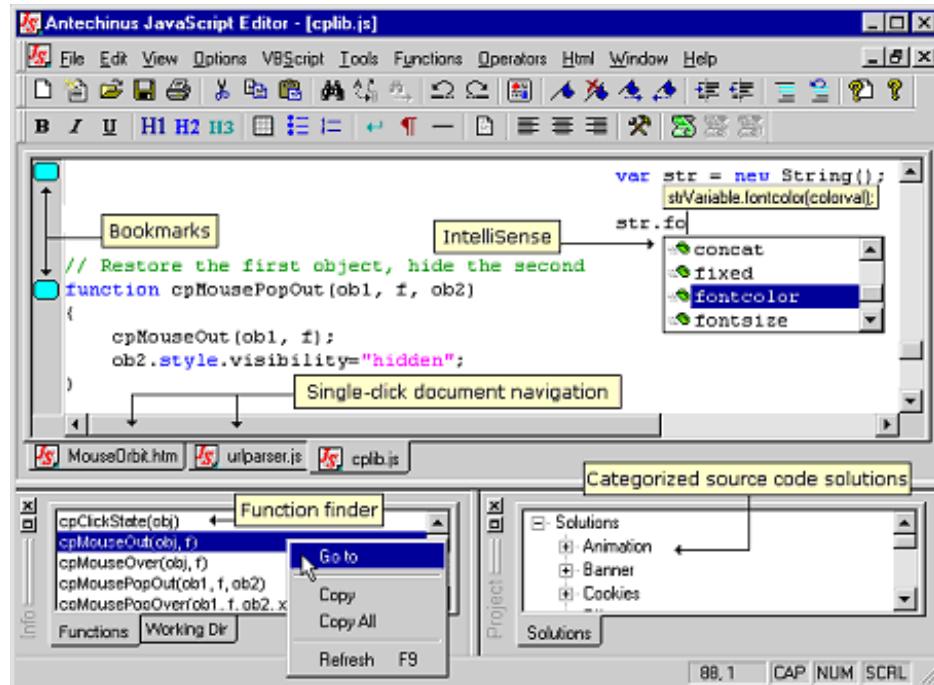


Figure 4. The Antechinus JavaScript editor.



# Index

## A

### Arrays

accessing elements .....	4-3
as structures.....	4-7
complex.....	4-7
concat method .....	4-19
declaring.....	4-2
defined.....	4-2
generateElements function .....	4-6
join method .....	4-19
length property .....	4-4, 4-18
multidimensional.....	4-15
new function.....	4-3
parallel arrays.....	4-12
simple arrays .....	4-5
slice method .....	4-21

## B

Boolean Object.....	6-21
Boolean operators.....	2-8
Browsers	
differences.....	1-8
event models .....	8-2
Browser support .....	1-6

## C

charCodeAt method.....	6-4
------------------------	-----

## D

Data Types.....	1-18
conversion .....	1-20
testing.....	1-21
Date	
constructors .....	6-12
Date object.....	6-8
Debugging .....	9-14
Venkman debugger .....	9-14
Document object.....	8-4
DOM .....	1-6, 1-9
hierarchy .....	1-9

## E

Error handling	
custom errors.....	9-9
Error messages	
displaying.....	9-2
interpreting .....	9-4
Errors.....	1-16
Evaluations.....	1-18

Event Handlers .....	1-11
Event Model .....	8-2
browser compatibility.....	8-21
browser differences .....	8-20
bubbling and IE4+.....	8-4
event capturing and Netscape Navigator 4 .....	8-7
event propagation.....	8-3
event sequence .....	8-2
events and Netscape Navigator 6+ .....	8-11
Event object.....	8-3
standard objects.....	8-15
static object .....	8-15
Event types .....	8-24
form events .....	8-26
keyboard events .....	8-25
mouse events .....	8-24
window events .....	8-25
Exception handling.....	9-6
Expressions .....	1-18

## F

Features .....	1-5
Form Objects .....	5-2
Forms	
action property .....	5-7
alternatives .....	5-2
Button objects .....	5-15
checkbox object .....	5-16
elements .....	5-10
event handlers .....	5-5
fieldset.....	5-8
file input object .....	5-26
form objects .....	5-2
method property .....	5-7
methods.....	5-4
onSubmit event .....	5-28
properties .....	5-4
radio object .....	5-17
select object.....	5-18
text object.....	5-11
validation .....	5-28
Functions .....	3-7
parameters .....	3-8
return value .....	3-9
syntax .....	3-7

## H

Hiding Script .....	1-15
History.....	1-2

## I

if, if else.....	2-2
iframe .....	7-19

## **J**

JScript..... 1-8

## **L**

Libraries ..... 1-16  
Loop structures ..... 2-11  
  do..while..... 2-14  
  for ..... 2-11  
  in ..... 2-14  
  while..... 2-13

## **M**

Math object..... 6-21  
  methods and properties..... 6-22  
  random numbers..... 6-22  
Math Objects ..... 6-19  
Methods..... 1-11

## **N**

Number object ..... 6-19  
  methods..... 6-21

## **O**

Operators  
  defined..... 1-19

## **P**

Placement ..... 1-13  
Properties..... 1-11

## **R**

Random numbers..... 6-22  
RegExp object ..... 6-24, 6-28  
  properties..... 6-29  
Regular Expressions ..... 6-24

## **S**

setInterval..... 6-14  
setTimeout..... 6-14  
Standards

ECMA..... 1-4

String  
  charAt method..... 6-3  
  fromCharCode ..... 6-5  
  match method..... 6-5  
  prototype ..... 6-2  
  replace method..... 6-6  
  split method..... 6-6  
String Object ..... 6-2  
Strings ..... 3-2  
  case changing..... 3-4  
  concatenation ..... 3-2  
  declaring ..... 3-2  
  substring extracting..... 3-5  
  substring search..... 3-4  
switch statement ..... 2-5

## **T**

Timer..... 6-14  
Try..catch.. finally blocks ..... 9-6

## **V**

Variables ..... 1-13, 1-18, 3-10  
  scope ..... 3-10  
Venkman debugger ..... 1-17  
vent types  
  load and unload ..... 8-25

## **W**

W3C ..... 1-6, 1-7, 1-8  
Window  
  attributes and browser support ..... 7-3  
  dynamic content ..... 7-5  
  frameset..... 7-13  
  functions ..... 7-8  
  iframe ..... 7-19  
  Modal dialogs ..... 7-12  
  open method..... 7-3  
  referencing ..... 7-5  
Window object ..... 7-2, 8-4  
With statement ..... 2-16

## **X**

XHTML ..... 1-13  
XML ..... 1-13