

AJAX Tutorial

AJAX is a web development technique for creating interactive web applications. If you know JavaScript, HTML, CSS, and XML, then you need to spend just one hour to start with AJAX.

Why to Learn Ajax?

AJAX stands for **A**synchronous **J**avaScript and **X**ML. AJAX is a new technique for creating better, faster, and more interactive web applications with the help of XML, HTML, CSS, and Java Script.

- Ajax uses XHTML for content, CSS for presentation, along with Document Object Model and JavaScript for dynamic content display.
- Conventional web applications transmit information to and from the sever using synchronous requests. It means you fill out a form, hit submit, and get directed to a new page with new information from the server.
- With AJAX, when you hit submit, JavaScript will make a request to the server, interpret the results, and update the current screen. In the purest sense, the user would never know that anything was even transmitted to the server.
- XML is commonly used as the format for receiving server data, although any format, including plain text, can be used.
- AJAX is a web browser technology independent of web server software.
- A user can continue to use the application while the client program requests information from the server in the background.
- Intuitive and natural user interaction. Clicking is not required, mouse movement is a sufficient event trigger.
- Data-driven as opposed to page-driven.

Rich Internet Application Technology

AJAX is the most viable Rich Internet Application (RIA) technology so far. It is getting tremendous industry momentum and several tool kit and frameworks are emerging. But at the same time, AJAX has browser incompatibility and it is supported by JavaScript, which is hard to maintain and debug.

AJAX is Based on Open Standards

AJAX is based on the following open standards –

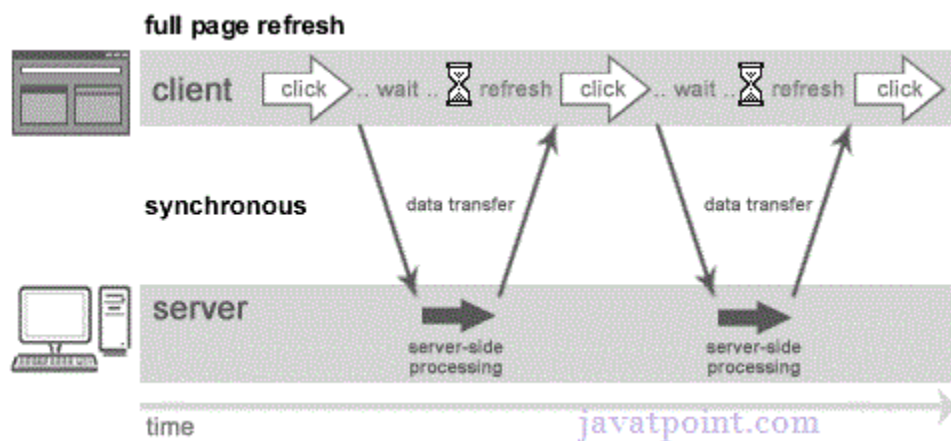
- Browser-based presentation using HTML and Cascading Style Sheets (CSS).
- Data is stored in XML format and fetched from the server.
- Behind-the-scenes data fetches using XMLHttpRequest objects in the browser.
- JavaScript to make everything happen.

Understanding Synchronous vs Asynchronous

Before understanding AJAX, let's understand classic web application model and ajax web application model first.

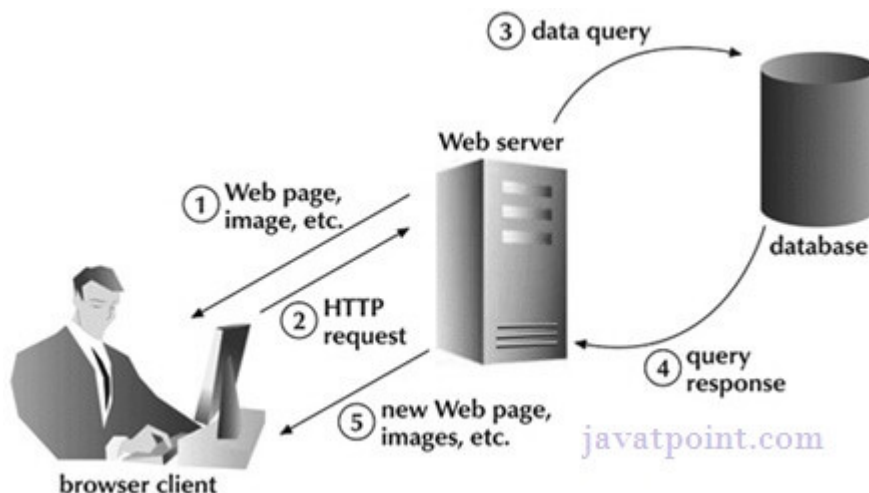
Synchronous (Classic Web-Application Model)

A synchronous request blocks the client until operation completes i.e. browser is unresponsive. In such case, javascript engine of the browser is blocked.



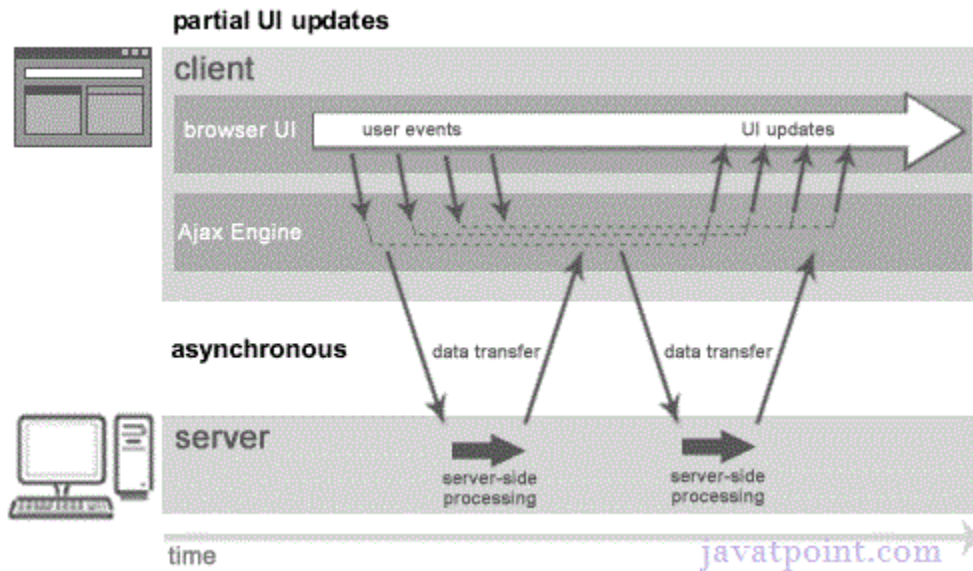
As you can see in the above image, full page is refreshed at request time and user is blocked until request completes.

Let's understand it another way.



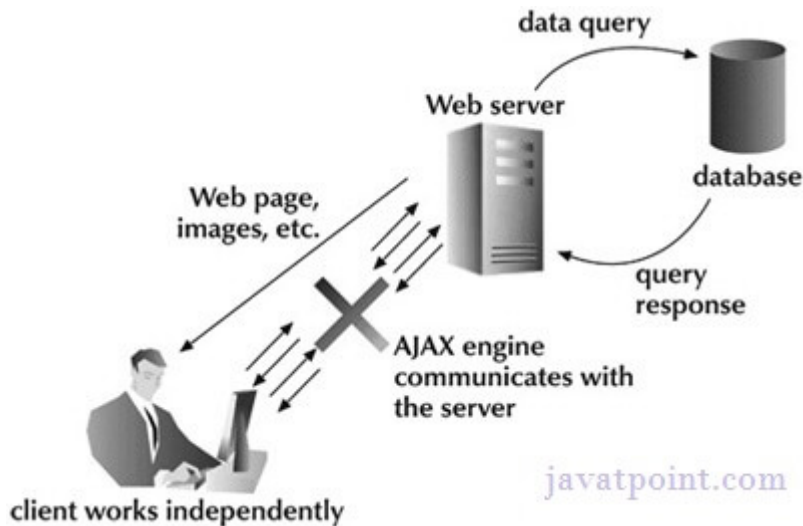
Asynchronous (AJAX Web-Application Model)

An asynchronous request doesn't block the client i.e. browser is responsive. At that time, user can perform another operations also. In such case, javascript engine of the browser is not blocked.



As you can see in the above image, full page is not refreshed at request time and user gets response from the ajax engine.

Let's try to understand asynchronous communication by the image given below.



AJAX Technologies

As describe earlier, ajax is not a technology but group of inter-related technologies. **AJAX** technologies includes:

- **HTML/XHTML** and **CSS**
- **DOM** (Document Object Model)
- **XML** or **JSON**
- **XMLHttpRequest**
- **JavaScript**

HTML/XHTML and CSS

These technologies are used for displaying content and style. It is mainly used for presentation.

DOM

It is used for dynamic display and interaction with data.

XML or JSON

For carrying data to and from server. JSON (Javascript Object Notation) is like XML but short and faster than XML.

XMLHttpRequest

For **asynchronous communication** between client and server.

JavaScript

It is used to bring above technologies together.

Independently, it is used mainly for client-side validation.

Understanding XMLHttpRequest

An object of XMLHttpRequest is used for asynchronous communication between client and server.

It performs following operations:

1. Sends data from the client in the background
2. Receives the data from the server
3. Updates the webpage without reloading it.

Properties of XMLHttpRequest object

The common properties of XMLHttpRequest object are as follows:

Property	Description
onReadyStateChange	It is called whenever readystate attribute changes. It must not be used with synchronous requests.
readyState	represents the state of the request. It ranges from 0 to 4. 0 UNOPENED open() is not called. 1 OPENED open is called but send() is not called. 2 HEADERS_RECEIVED send() is called, and headers and status are available. 3 LOADING Downloading data; responseText holds the data. 4 DONE The operation is completed fully.
responseText	returns response as text.
responseXML	returns response as XML

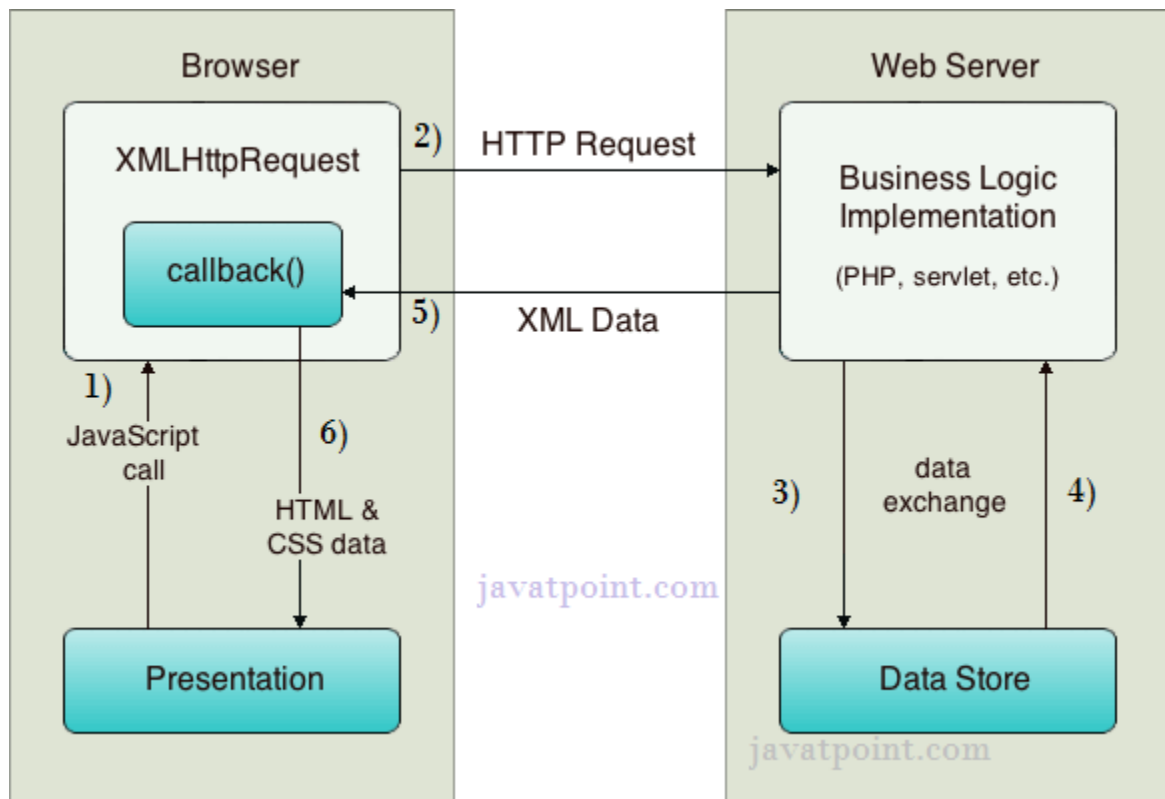
Methods of XMLHttpRequest object

The important methods of XMLHttpRequest object are as follows:

Method	Description
void open(method, URL)	opens the request specifying get or post method and url.
void open(method, URL, async)	same as above but specifies asynchronous or not.
void open(method, URL, async, username, password)	same as above but specifies username and password.
void send()	sends get request.
void send(string)	send post request.
setRequestHeader(header,value)	it adds request headers.

How AJAX works?

AJAX communicates with the server using XMLHttpRequest object. Let's try to understand the flow of ajax or how ajax works by the image displayed below.



As you can see in the above example, XMLHttpRequest object plays a important role.

1. User sends a request from the UI and a javascript call goes to XMLHttpRequest object.
2. HTTP Request is sent to the server by XMLHttpRequest object.
3. Server interacts with the database using JSP, PHP, Servlet, ASP.net etc.
4. Data is retrieved.
5. Server sends XML data or JSON data to the XMLHttpRequest callback function.
6. HTML and CSS data is displayed on the browser.

JavaScript Trends in 2021

2022 is just around the corner, as unbelievable as that sounds. If you're curious about what the future of the programming world might be, you're in the right place. We tried to analyze trends in 2022 and want to share some insights into key directions for the JavaScript ecosystem.

Read on to find out!

JavaScript Language Keeps Going Strong

For years in a row, JavaScript is the most sought-after and fast-growing programming language. It remains one of the smartest choices when it comes to the development of interactive web interfaces since it's supported by all modern browsers.

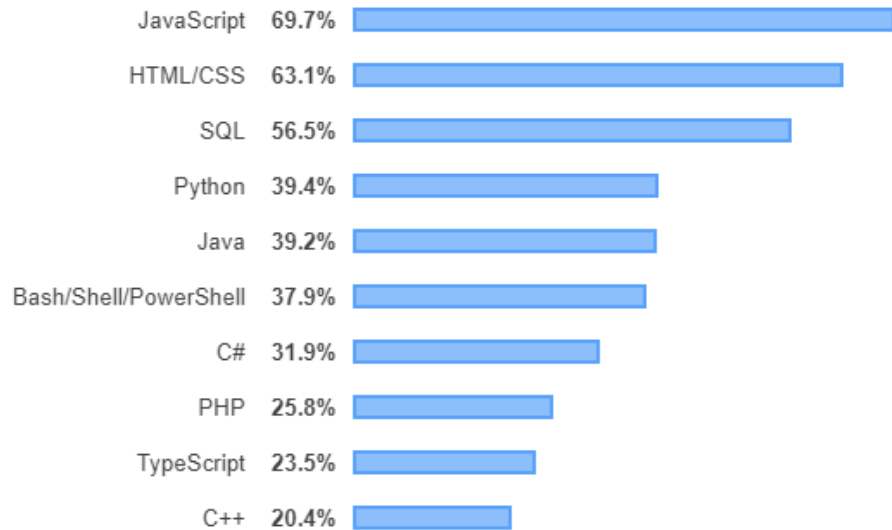
As the [annual survey held by Stack Overflow](#) shows, about 70 percent of 72.525 professional developers stated they use JavaScript. Moreover, it's one of the most wanted languages meaning that 17.8% of respondents have not yet used it but want to learn it.



Programming, Scripting, and Markup Languages

All Respondents

Professional Developers



According to the data [provided by stackshare.io](https://stackshare.io), over 10400 companies worldwide use JavaScript in their stacks. The language is the heart of any big tech company, such as PayPal (likewise, the online payment giant was one of the earliest adopters of NodeJS), Netflix, Groupon, Walmart, and LinkedIn.



After all, [16 from 25 US unicorn companies](#) (the top privately-held startups valued at over \$1 billion) mention JavaScript in their technology stacks. It's therefore unlikely that JavaScript goes off the grid in the near future.

JavaScript Best Practices

Avoid Global Variables

Minimize the use of global variables.

This includes all data types, objects, and functions.

Global variables and functions can be overwritten by other scripts.

Use local variables instead

Always Declare Local Variables

All variables used in a function should be declared as **local** variables.

Local variables **must** be declared with the **var** keyword or the **let** keyword, otherwise they will become global variables.

Differences between let, var and const (Javascript)

	var	let	const
scope	global or function-based	block	block
re-declaration	can be re-declared	cannot be re-declared	cannot be re-declared
update value	value can be updated	value can be updated	value cannot be updated

faithgaiciumia.hashnode.dev

Declarations on Top

It is a good coding practice to put all declarations at the top of each script or function.

This will:

- Give cleaner code
- Provide a single place to look for local variables
- Make it easier to avoid unwanted (implied) global variables
- Reduce the possibility of unwanted re-declarations

```
// Declare at the beginning
let firstName, lastName, price, discount, fullPrice;

// Use later
firstName = "John";
lastName = "Doe";

price = 19.90;
discount = 0.10;

fullPrice = price - discount;
```

This also goes for loop variables:

```
for (let i = 0; i < 5; i++) {
```

Initialize Variables

It is a good coding practice to initialize variables when you declare them.

This will:

- Give cleaner code
- Provide a single place to initialize variables
- Avoid undefined values

```
// Declare and initiate at the beginning
let firstName = "",
let lastName = "",
let price = 0,
let discount = 0,
let fullPrice = 0,
const myArray = [],
const myObject = {};
```

Initializing variables provides an idea of the intended use (and intended data type).

Declare Objects with **const**

Declaring objects with `const` will prevent any accidental change of type:

Example

```
let car = {type:"Fiat", model:"500", color:"white"};
car = "Fiat"; // Changes object to string
```

```
const car = {type:"Fiat", model:"500", color:"white"};
car = "Fiat"; // Not possible
```

Declare Arrays with **const**

Declaring arrays with `const` will prevent any accidental change of type:

Example

```
let cars = ["Saab", "Volvo", "BMW"];  
cars = 3;    // Changes array to number
```

```
const cars = ["Saab", "Volvo", "BMW"];  
cars = 3;    // Not possible
```

Don't Use `new Object()`

- Use `""` instead of `new String()`
- Use `0` instead of `new Number()`
- Use `false` instead of `new Boolean()`
- Use `{}` instead of `new Object()`
- Use `[]` instead of `new Array()`
- Use `/()/` instead of `new RegExp()`
- Use `function (){}` instead of `new Function()`

Example

```
let x1 = "";           // new primitive string  
let x2 = 0;           // new primitive number  
let x3 = false;       // new primitive boolean  
const x4 = {};        // new object  
const x5 = [];        // new array object  
const x6 = /()/;      // new regexp object  
const x7 = function(){}; // new function object
```

Beware of Automatic Type Conversions

JavaScript is loosely typed.

A variable can contain all data types.

A variable can change its data type:

Example

```
let x = "Hello";    // typeof x is a string
x = 5;             // changes typeof x to a number
```

[Try it Yourself »](#)

Beware that numbers can accidentally be converted to strings or **NaN** (Not a Number).

When doing mathematical operations, JavaScript can convert numbers to strings:

Example

```
let x = 5 + 7;      // x.valueOf() is 12, typeof x is a number
let x = 5 + "7";   // x.valueOf() is 57, typeof x is a string
let x = "5" + 7;   // x.valueOf() is 57, typeof x is a string
let x = 5 - 7;     // x.valueOf() is -2, typeof x is a number
let x = 5 - "7";   // x.valueOf() is -2, typeof x is a number
let x = "5" - 7;   // x.valueOf() is -2, typeof x is a number
let x = 5 - "x";   // x.valueOf() is NaN, typeof x is a number
```

[Try it Yourself »](#)

Subtracting a string from a string, does not generate an error but returns **NaN** (Not a Number):

Example

```
"Hello" - "Dolly" // returns NaN
```

[Try it Yourself »](#)

Use === Comparison

The `==` comparison operator always converts (to matching types) before comparison.

The `===` operator forces comparison of values and type:

Example

```
0 == "";           // true
1 == "1";          // true
1 == true;         // true

0 === "";          // false
1 === "1";         // false
1 === true;        // false
```

[Try it Yourself »](#)

Use Parameter Defaults

If a function is called with a missing argument, the value of the missing argument is set to `undefined`.

Undefined values can break your code. It is a good habit to assign default values to arguments.

Example

```
function myFunction(x, y) {
  if (y === undefined) {
    y = 0;
  }
}
```

[Try it Yourself »](#)

End Your Switches with Defaults

Always end your `switch` statements with a `default`. Even if you think there is no need for it.

Example

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
```

```
    day = "Tuesday";
    break;
case 3:
    day = "Wednesday";
    break;
case 4:
    day = "Thursday";
    break;
case 5:
    day = "Friday";
    break;
case 6:
    day = "Saturday";
    break;
default:
    day = "Unknown";
}
```

[Try it Yourself »](#)

Avoid Number, String, and Boolean as Objects

Always treat numbers, strings, or booleans as primitive values. Not as objects.

Declaring these types as objects, slows down execution speed, and produces nasty side effects:

Example

```
let x = "John";
let y = new String("John");
(x === y) // is false because x is a string and y is an object.
```

[Try it Yourself »](#)

Or even worse:

Example

```
let x = new String("John");
let y = new String("John");
(x == y) // is false because you cannot compare objects.
```


JavaScript Common Mistakes(Quality code)

Accidentally Using the Assignment Operator

JavaScript programs may generate unexpected results if a programmer accidentally uses an assignment operator (=), instead of a comparison operator (==) in an if statement.

This `if` statement returns `false` (as expected) because `x` is not equal to `10`:

```
let x = 0;  
if (x == 10)
```

[Try it Yourself »](#)

This `if` statement returns `true` (maybe not as expected), because `10` is true:

```
let x = 0;  
if (x = 10)
```

[Try it Yourself »](#)

This `if` statement returns `false` (maybe not as expected), because `0` is false:

```
let x = 0;  
if (x = 0)
```

[Try it Yourself »](#)

An assignment always returns the value of the assignment.

Expecting Loose Comparison

In regular comparison, data type does not matter. This `if` statement returns true:

```
let x = 10;
let y = "10";
if (x == y)
```

[Try it Yourself »](#)

In strict comparison, data type does matter. This `if` statement returns false:

```
let x = 10;
let y = "10";
if (x === y)
```

[Try it Yourself »](#)

It is a common mistake to forget that `switch` statements use strict comparison:

This `case switch` will display an alert:

```
let x = 10;
switch(x) {
  case 10: alert("Hello");
}
```

[Try it Yourself »](#)

This `case switch` will not display an alert:

```
let x = 10;
switch(x) {
  case "10": alert("Hello");
}
```

[Try it Yourself »](#)

Confusing Addition & Concatenation

Addition is about adding **numbers**.

Concatenation is about adding **strings**.

In JavaScript both operations use the same `+` operator.

Because of this, adding a number as a number will produce a different result from adding a number as a string:

```
let x = 10;
x = 10 + 5;      // Now x is 15

let y = 10;
y += "5";       // Now y is "105"
```

[Try it Yourself »](#)

When adding two variables, it can be difficult to anticipate the result:

```
let x = 10;
let y = 5;
let z = x + y;   // Now z is 15

let x = 10;
let y = "5";
let z = x + y;   // Now z is "105"
```

[Try it Yourself »](#)

Misunderstanding Floats

All numbers in JavaScript are stored as 64-bits **Floating point numbers** (Floats).

All programming languages, including JavaScript, have difficulties with precise floating point values:

```
let x = 0.1;
let y = 0.2;
let z = x + y      // the result in z will not be 0.3
```

[Try it Yourself »](#)

To solve the problem above, it helps to multiply and divide:

Example

```
let z = (x * 10 + y * 10) / 10;    // z will be 0.3
```

[Try it Yourself »](#)

Breaking a JavaScript String

JavaScript will allow you to break a statement into two lines:

Example 1

```
let x =  
"Hello World!";
```

[Try it Yourself »](#)

But, breaking a statement in the middle of a string will not work:

Example 2

```
let x = "Hello  
World!";
```

[Try it Yourself »](#)

You must use a "backslash" if you must break a statement in a string:

Example 3

```
let x = "Hello \  
World!";
```

[Try it Yourself »](#)

Misplacing Semicolon

Because of a misplaced semicolon, this code block will execute regardless of the value of x:

```
if (x == 19);  
{  
  // code block  
}
```

[Try it Yourself »](#)

Breaking a Return Statement

It is a default JavaScript behavior to close a statement automatically at the end of a line.

Because of this, these two examples will return the same result:

Example 1

```
function myFunction(a) {  
  let power = 10  
  return a * power  
}
```

[Try it Yourself »](#)

Example 2

```
function myFunction(a) {  
  let power = 10;  
  return a * power;  
}
```

[Try it Yourself »](#)

JavaScript will also allow you to break a statement into two lines.

Because of this, example 3 will also return the same result:

Example 3

```
function myFunction(a) {  
  let  
  power = 10;  
  return a * power;  
}
```

[Try it Yourself »](#)

But, what will happen if you break the return statement in two lines like this:

Example 4

```
function myFunction(a) {  
  let
```

```
power = 10;  
return  
a * power;  
}
```

[Try it Yourself »](#)

The function will return `undefined`!

Why? Because JavaScript thought you meant:

Example 5

```
function myFunction(a) {  
  let  
  power = 10;  
  return;  
  a * power;  
}
```

[Try it Yourself »](#)

Explanation

If a statement is incomplete like:

```
let
```

JavaScript will try to complete the statement by reading the next line:

```
power = 10;
```

But since this statement is complete:

```
return
```

JavaScript will automatically close it like this:

```
return;
```

This happens because closing (ending) statements with semicolon is optional in JavaScript.

JavaScript will close the return statement at the end of the line, because it is a complete statement.

Never break a return statement.

Accessing Arrays with Named Indexes

Many programming languages support arrays with named indexes.

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.

In JavaScript, **arrays** use **numbered indexes**:

Example

```
const person = [];  
person[0] = "John";  
person[1] = "Doe";  
person[2] = 46;  
person.length;           // person.length will return 3  
person[0];               // person[0] will return "John"
```

[Try it Yourself »](#)

In JavaScript, **objects** use **named indexes**.

If you use a named index, when accessing an array, JavaScript will redefine the array to a standard object.

After the automatic redefinition, array methods and properties will produce undefined or incorrect results:

Example:

```
const person = [];  
person["firstName"] = "John";  
person["lastName"] = "Doe";  
person["age"] = 46;  
person.length;           // person.length will return 0  
person[0];               // person[0] will return undefined
```

[Try it Yourself »](#)

Ending Definitions with a Comma

Trailing commas in object and array definition are legal in ECMAScript 5.

Object Example:

```
person = {firstName:"John", lastName:"Doe", age:46,}
```

Array Example:

```
points = [40, 100, 1, 5, 25, 10,];
```

Undefined is Not Null

JavaScript objects, variables, properties, and methods can be `undefined`.

In addition, empty JavaScript objects can have the value `null`.

This can make it a little bit difficult to test if an object is empty.

You can test if an object exists by testing if the type is `undefined`:

Example:

```
if (typeof myObj === "undefined")
```

[Try it Yourself »](#)

But you cannot test if an object is `null`, because this will throw an error if the object is `undefined`:

Incorrect:

```
if (myObj === null)
```

To solve this problem, you must test if an object is not `null`, and not `undefined`.

But this can still throw an error:

Incorrect:

```
if (myObj !== null && typeof myObj !== "undefined")
```


Because of this, you must test for not `undefined` before you can test for not `null`:

Correct:

```
if (typeof myObj !== "undefined" &&myObj !== null)
```

[Try it Yourself »](#)

JavaScript Form

In this tutorial, we will learn, discuss, and understand the JavaScript form. We will also see the implementation of the JavaScript form for different purposes.

Here, we will learn the method to access the form, getting elements as the JavaScript form's value, and submitting the form.

Introduction to Forms

Forms are the basics of [HTML](#)

. We use [HTML form element](#) in order to create the [JavaScript](#) form. For creating a form, we can use the following sample code:

```
1. <html>
2. <head>
3. <title> Login Form</title>
4. </head>
5. <body>
6. <h3> LOGIN </h3>
7. <form name="Login_form" onsubmit="submit_form()">
8. <h4> USERNAME</h4>
9. <input type="text" placeholder="Enter your email id"/>
10. <h4> PASSWORD</h4>
11. <input type="password" placeholder="Enter your password"/><br></br>
12. <input type="submit" value="Login"/>
13. <input type="button" value="SignUp" onClick="create()"/>
14. </form>
15. </html>
```

In the code:

- Form name tag is used to define the name of the form. The name of the form here is "Login_form". This name will be referenced in the JavaScript form.
- The action tag defines the action, and the browser will take to tackle the form when it is submitted. Here, we have taken no action.
- The method to take action can be either **post** or **get**, which is used when the form is to be submitted to the server. Both types of methods have their own properties and rules.
- The input type tag defines the type of inputs we want to create in our form. Here, we have used input type as 'text', which means we will input values as text in the textbox.
- Net, we have taken input type as 'password' and the input value will be password.

- Next, we have taken input type as 'button' where on clicking, we get the value of the form and get displayed.

Other than action and methods, there are the following useful methods also which are provided by the HTML Form Element

- **submit ()**: The method is used to submit the form.
- **reset ()**: The method is used to reset the form values.

Referencing forms

Now, we have created the form element using HTML, but we also need to make its connectivity to JavaScript. For this, we use **the getElementById ()** method that references the html form element to the JavaScript code.

The syntax of using the **getElementById()** method

is as follows:

1. let `form` = `document.getElementById('subscribe');`

Using the Id, we can make the reference.

Submitting the form

Next, we need to submit the form by submitting its value, for which we use the **onSubmit()** method. Generally, to submit, we use a submit button that submits the value entered in the form.

The syntax of the submit() method is as follows:

1. `<input type="submit" value="Subscribe">`

When we submit the form, the action is taken just before the request is sent to the server. It allows us to add an event listener that enables us to place various validations on the form. Finally, the form gets ready with a combination of HTML and JavaScript code.

Let's collect and use all these to create a **Login form**

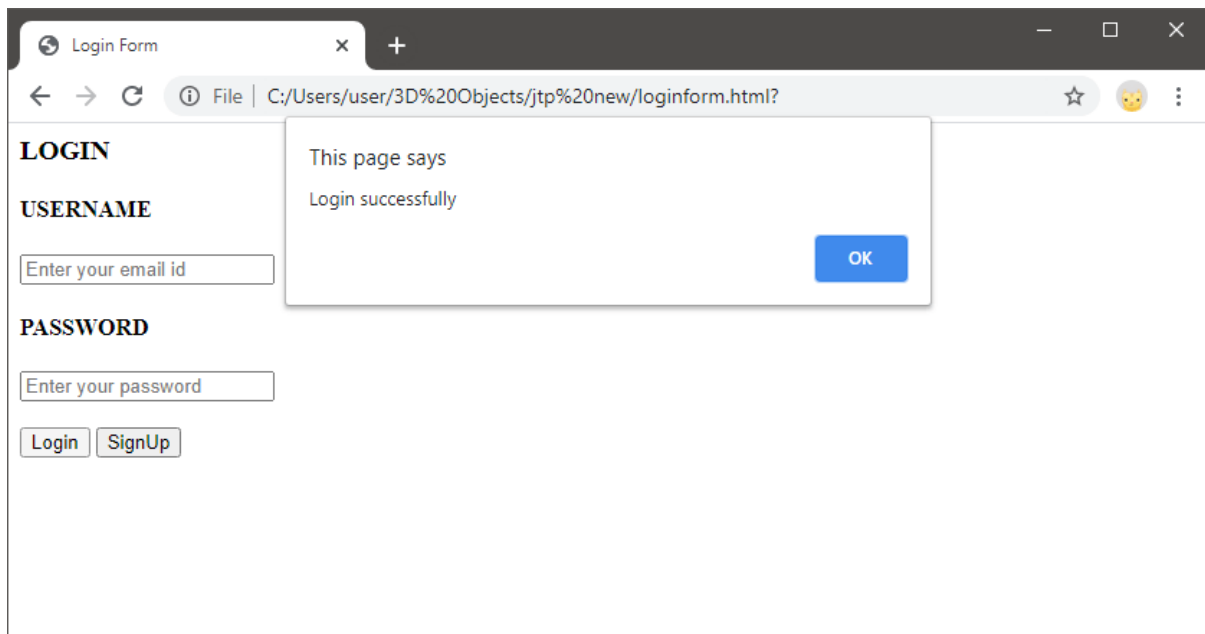
and **SignUp form** and use both.

Login Form

1. `<html>`
2. `<head>`
3. `<title> Login Form</title>`
4. `</head>`
5. `<body>`

6. `<h3> LOGIN </h3>`
7. `<form name="Login_form" onsubmit="submit_form()">`
8. `<h4> USERNAME</h4>`
9. `<input type="text" placeholder="Enter your email id"/>`
10. `<h4> PASSWORD</h4>`
11. `<input type="password" placeholder="Enter your password"/></br></br>`
12. `<input type="submit" value="Login"/>`
13. `<input type="button" value="SignUp" onClick="create()"/>`
14. `</form>`
15. `<script type="text/javascript">`
16. `function submit_form(){`
17. `alert("Login successfully");`
18. `}`
19. `function create(){`
20. `window.location="signup.html";`
21. `}`
22. `</script>`
23. `</body>`
24. `</html>`

The output of the above code on clicking on Login button is shown below:



SignUp Form

1. `<html>`
2. `<head>`
3. `<title> SignUp Page</title>`
4. `</head>`
5. `<body align="center" >`
6. `<h1> CREATE YOUR ACCOUNT</h1>`

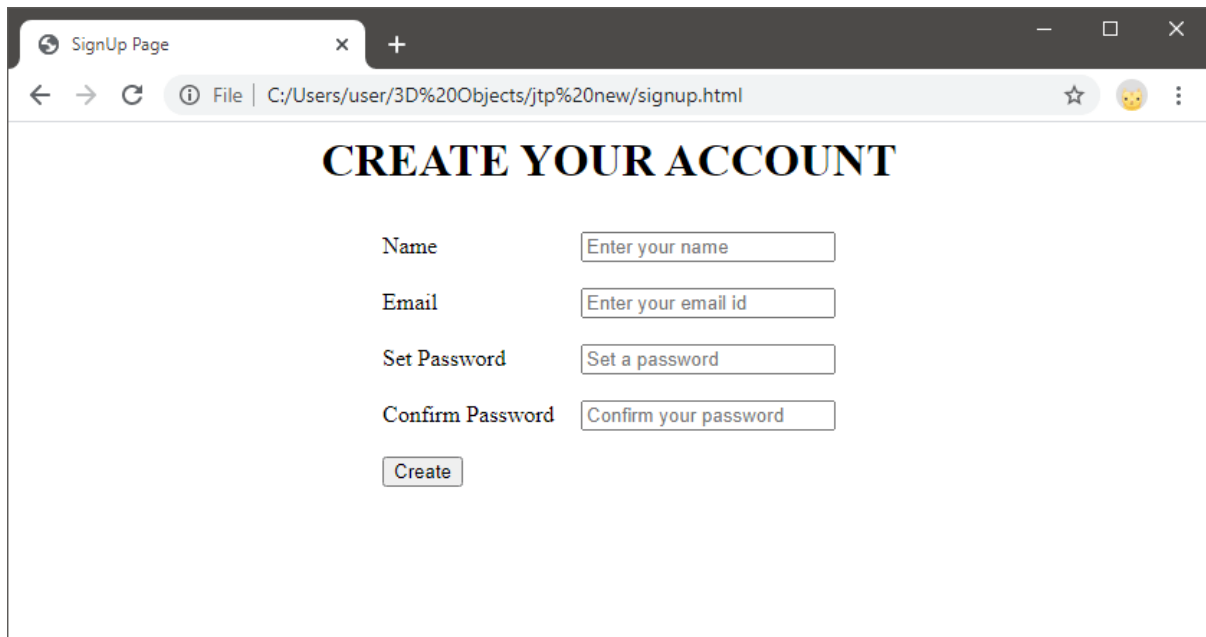
```

7. <table cellpadding="2" cellspacing="2" align="center" border="1">
8. <tr><td> Name</td>
9. <td><input type="text" placeholder="Enter your name" id="n1"></td></tr>
10. <tr><td>Email </td>
11. <td><input type="text" placeholder="Enter your email id" id="e1"></td></tr>
12. <tr><td> Set Password</td>
13. <td><input type="password" placeholder="Set a password" id="p1"></td></tr>
14. <tr><td>Confirm Password</td>
15. <td><input type="password" placeholder="Confirm your password" id="p2"></td>
    </tr>
16. <tr><td>
17. <input type="submit" value="Create" onClick="create_account()" />
18. </td></tr>
19. </table>
20. <script type="text/javascript">
21. function create_account(){
22.     var n=document.getElementById("n1").value;
23.     var e=document.getElementById("e1").value;
24.     var p=document.getElementById("p1").value;
25.     var cp=document.getElementById("p2").value;
26.     //Code for password validation
27.     var letters = /^[A-Za-z]+$;/
28.     var email_val = /^[a-zA-Z0-9_\.\-]+\@((\[[a-zA-Z0-9\-\]]+\.)+|[a-zA-Z0-9-
29.     9]{2,4})+$/;
30.     //other validations required code
31.     if(n=="||e=="||p=="||cp==" ){
32.         alert("Enter each details correctly");
33.     }
34.     else if(!letters.test(n))
35.     {
36.         alert('Name is incorrect must contain alphabets only');
37.     }
38.     else if (!email_val.test(e))
39.     {
40.         alert('Invalid email format please enter valid email id');
41.     }
42.     else if(p!=cp)
43.     {
44.         alert("Passwords not matching");
45.     }
46.     else if(document.getElementById("p1").value.length > 12)
47.     {
48.         alert("Password maximum length is 12");
49.     }
50.     else if(document.getElementById("p1").value.length < 6)
51.     {
52.         alert("Password minimum length is 6");
53.     }
54.     else{

```

```
53. alert("Your account has been created successfully... Redirecting to JavaTpoint.co  
m");  
54. window.location="https://www.Facebook.com/";  
55. }  
56. }  
57. </script>  
58. </body>  
59. </html>
```

The output of the above code is shown below:



The screenshot shows a web browser window titled "SignUp Page" with the address bar displaying "File | C:/Users/user/3D%20Objects/jtp%20new/signup.html". The main content of the page is a form titled "CREATE YOUR ACCOUNT" in a large, bold, serif font. Below the title, there are four rows of input fields, each with a label to its left and a text box to its right. The labels are "Name", "Email", "Set Password", and "Confirm Password". The text boxes contain the placeholder text "Enter your name", "Enter your email id", "Set a password", and "Confirm your password" respectively. Below these fields is a single "Create" button.

In this way, we can create forms in JavaScript with proper validations.

JavaScript Browser Object Model

JavaScript provides WebAPIs and Interfaces(object types) that we can use while developing web application or website. These APIs and objects help us in controlling the lifecycle of the webpage and performing various actions like getting browser information, managing screen size, opening and closing new browser window, getting URL information or updating URL, getting cookies and local storage, etc.

The Interfaces (object types) which help us **interact with the browser window** are known as **Browser objects**. Browser object is not an official term but its a group of objects which belongs to different WebAPIs but are used for managing various browser related information and actions.

For example, when an HTML document is opened in a browser window, the browser interprets the document as a collection of hierarchical objects(HTML tags) and accordingly displays the data contained in these objects(HTML page rendering). The browser parses the document and creates a collection of objects, which defines the documents and its details. We have shown the various objects that can be used to access various components of the browser window in the picture below:



The browser objects are of various types, used for interacting with the browser and belongs to different APIs. The collection of these Browser objects is also known as **Browser object Model(BOM)**.

The default object of browser is Window which means you can call its functions directly.

Browser Objects:

The objects listed below are called browser objects.

- Window - part of DOM API
- Navigator

- Document – part of DOM API
- Screen – property of Window object
- History – property of Window object
- Location – property of Window and Document object

Window Object

It is used to interact with the browser window which displays the web page. It generally **represents a tab in browser**, but some actions like window width and height will affect the complete browser window.

We have covered JavaScript Window Object separately, in detail.

Navigator Object

It acts as a storehouse of all the data and **information about the Browser software used to access the webpage** and this object is used to fetch information related to the browser for example, whether the user is using Chrome browser or Safari browser, which version of browser is being used etc.

We have covered JavaScript Navigator Object separately, in detail.

Document Object

This object **represent the HTML document** that is loaded into the browser. A document object is an object that provides access to all HTML elements of a document(webpage). We can use this object to append a new HTML tag to the webpage, modify any existing HTML tag, etc.

We have covered JavaScript Document Object separately, in detail.

History Object

It stores Uniform Resource Locator(URLs) visited by a user in the browser. It is a built-in object which is used to get browser history. This object is a property of the JavaScript Window object.

We have covered JavaScript History Object separately, in detail.

Screen Object

It is a built-in object which is used to **fetch information related to the browser screen**, like the screen size, etc. It is also obtained from the Window object.

We have covered JavaScript Screen Object separately, in detail.

Location Object

Location is a built-in object which **represent the location of the object to which it is linked**, which can be Window or Document. Both the Document and Window interface have a linked location property.

JavaScript - Multimedia

The JavaScript **navigator** object includes a child object called **plugins**. This object is an array, with one entry for each plug-in installed on the browser. The navigator.plugins object is supported only by Netscape, Firefox, and Mozilla only.

Example

Here is an example that shows how to list down all the plug-on installed in your browser –

```
<html>
<head>
<title>List of Plug-Ins</title>
</head>

<body>
<tableborder="1">
<tr>
<th>Plug-in Name</th>
<th>Filename</th>
<th>Description</th>
</tr>

<scriptlanguage="JavaScript"type="text/javascript">
for(i=0;i<navigator.plugins.length;i++){
document.write("<tr><td>");
document.write(navigator.plugins[i].name);
document.write("</td><td>");
document.write(navigator.plugins[i].filename);
document.write("</td><td>");
```

```
document.write(navigator.plugins[i].description);
document.write("</td></tr>");
}
</script>
</table>
</body>
</html>
```

Output

Plug-in Name	Filename	Description
PDF Viewer	internal-pdf-viewer	Portable Document Format
Chrome PDF Viewer	internal-pdf-viewer	Portable Document Format
Chromium PDF Viewer	internal-pdf-viewer	Portable Document Format
Microsoft Edge PDF Viewer	internal-pdf-viewer	Portable Document Format
WebKit built-in PDF	internal-pdf-viewer	Portable Document Format

Controlling Multimedia

Let us take one real example which works in almost all the browsers –

```
<html>
<head>
<title>Using Embedded Object</title>

<scripttype="text/javascript">
<!--
functionplay(){
```

```
if(!document.demo.IsPlaying()){
document.demo.Play();
}
}
functionstop(){
if(document.demo.IsPlaying()){
document.demo.StopPlay();
}
}
functionrewind(){
if(document.demo.IsPlaying()){
document.demo.StopPlay();
}
document.demo.Rewind();
}
//-->
</script>
</head>

<body>
<embedid="demo"name="demo"
src="http://www.amrood.com/games/kumite.swf"
width="318"height="300"play="false"loop="false"
pluginspage="http://www.macromedia.com/go/getflashplayer"
swliveconnect="true">

<formname="form"id="form"action="#"method="get">
<inputtype="button"value="Start"onclick="play();"/>
<inputtype="button"value="Stop"onclick="stop();"/>
<inputtype="button"value="Rewind"onclick="rewind();"/>
</form>
</body>
</html>
```

JavaScript The Complete Reference (2012)

PART III Applied JavaScript

CHAPTER 3 Windows, Frames, and Overlays

Now it is time to begin to put the syntax and theory we have covered up to this point in the book to use. Starting from the top of the object hierarchy with `Window`, we will explore some applications of JavaScript. In this chapter, we will learn how to create a variety of windows, including alerts, confirmations, prompts, and custom pop-up windows of our own design. We will also show how windows and frames are very much related. Finally, we'll discuss the problematic special cases of window management and how overlays have become a needed tool in the JavaScript developer's toolbox.

Introduction to the window Object

JavaScript's `Window` object represents the browser window, or potentially frame, that a document is displayed in. The properties of a particular instance of `Window` might include its size, amount of chrome—namely the buttons, scroll bars, and so on—in the browser frame, position, and so on. The methods of the `Window` include the creation and destruction of generic windows and the handling of special-case windows such as `alert`, `confirmation`, and `prompt` dialogs. The `Window`, at least in the case of browser-based JavaScript, defines the universe, and most everything lives within it. As the topmost object in the JavaScript browser-document object hierarchy, `Window` contains references to any of the DOM or browser-related objects we have presented to this point, as well as any user-defined global values; thus it seems appropriate to start our discussion of applied JavaScript with it.

As we have discussed numerous times in the book, the `Window` is not only home to many useful properties, methods, and objects but to the variables and functions that we define in our scripts. For example, if we define a variable *likeJavaScript* and set it to true,

```
var likeJavaScript = true;
```

and we are not within function scope, that variable is global and, as such, becomes a property of the Window object. In other words, if we write code like this:

```
alert(window.likeJavaScript); // true
```

it is the same as if we were to write this code:

```
alert(likeJavaScript); // true
```

Yet, even more interesting, the alert() method itself is part of the Window, so

```
window.alert(window.likeJavaScript);
```

is also the same thing.

As we have discussed earlier, we must be quite careful not to collide with other scripts' global variables. Thus we would tend to use a wrapper object to house our variables, therefore limiting our exposure to the global name space as much as possible:

```
var JSREF = {};  
JSREF.likeJavaScript = true;
```

It turns out that such a scheme of limiting our identifier footprint isn't just useful to avoid clashing with other included scripts, but to avoid clashing with the properties, methods, and objects of the Window object itself.

Table 12-1 shows the properties, including objects, of the Window object, while *Table 12-2* shows its methods. The tables contain data collected primarily from the HTML5 specification; however, they also include proprietary properties and methods that have spread across multiple browsers or are often seen in real-world code bases. These tables should provide a useful roadmap to the Window object.

Table 12-1 window Properties and Objects

Property	Description
ActiveXObject	Internet Explorer 9 returns a null value and says not to use it; however, previous Explorer versions do support this object.
applicationCache	Returns the application cache object for the window.
clientInformation	Object containing information about the user's browser and operating system.
closed	Boolean indicating if the Window object is closed.
constructor	Reference to the object's constructor.
content	Reference to the topmost Window object.
defaultStatus	The default message in the status bar.
dialogArguments	The argument(s) passed into <code>showModalDialog()</code> or <code>showModelessDialog()</code> .
dialogHeight	The height of a dialog window.
dialogLeft	The left position of a dialog window.
dialogTop	The top position of a dialog window.
dialogWidth	The width of a dialog window.
directories	This property is obsolete and is replaced by <code>personalbar</code> .
document	Reference to the Document object that allows manipulation of the elements in the page.

Property	Description
event	Object that holds information about the current event.
external	Reference to the external object that contains additional functionalities. The external object has many functions in Internet Explorer. Now the specification includes this object, though currently it only defines OpenSearch methods.
frameElement	The frame that the current window is embedded in.
frames[]	Array of frames in the page.
fullScreen	Boolean indicating if the browser is in full-screen mode.
globalStorage	Reference to a storage object that is used to store information across pages.
history	Reference to the History object that allows access to some data in the user's history, as well as methods for navigating the history.
Image	Creates a new Image object and returns a reference to the object.
innerHeight	The browser's client area height, including scroll bars.
innerWidth	The browser's client area width, including scroll bars.
length	The number of frames in the current page.
location	Reference to the Location object that returns information about the current URL and provides methods for manipulating the URL.

locationbar	Reference to a <code>BarProp</code> representing the location bar.
localStorage	Reference to a storage object that is used to store information across pages.
menubar	Reference to a <code>BarProp</code> representing the menu bar.
name	The name of the window.
navigator	Reference to the <code>Navigator</code> object that provides information about the user's browser and operating system.
opener	Reference to the <code>Window</code> object that opened the current window.
Option	Creates a new <code>Option</code> object and returns a reference to the object.
outerHeight	The height of the entire browser window, including the toolbars.
outerWidth	The width of the entire browser window, including the toolbars.
pageXOffset	The number of pixels that the page is scrolled to the left.
pageYOffset	The number of pixels that the page is scrolled down.
parent	Reference to the <code>Window</code> object that is the parent of the current window.
performance	Reference to a <code>Performance</code> object found at <code>window.performance</code> that contains data about the performance and timing of a Web page load.
personalbar	Reference to a <code>BarProp</code> representing the bookmark bar.

returnValue	The return value to be sent back to the calling function after <code>window.showModalDialog()</code> is called.
screen	Reference to the <code>Screen</code> object that contains information about the user's screen.
screenLeft	The x position of the top-left corner of the browser's client area.
screenTop	The y position of the top-left corner of the browser's client area.
screenX	The x position of the top-left corner of the browser window.
screenY	The x position of the top-left corner of the browser window.
scrollbars	Reference to a <code>BarProp</code> representing the scroll bars.
scrollX	The number of pixels the page is scrolled horizontally.
scrollY	The number of pixels the page is scrolled vertically.
self	Reference to the current <code>Window</code> object.
sessionStorage	Reference to a storage object that is used to store information in a single session.
sidebar	Reference to the sidebar object that allows some manipulation of the sidebar.
status	The message in the status bar.
statusbar	Reference to a <code>BarProp</code> representing the status bar.
toolbar	Reference to a <code>BarProp</code> representing the toolbar.
top	Reference to the topmost <code>Window</code> object.
URL	Reference to the <code>URL</code> object that provides methods for creating object URLs.
window	Reference to the current window.
XDomainRequest	Creates and returns an <code>XDomainRequest</code> object, which provides functionality for cross-domain Ajax requests.
XMLHttpRequest	Creates and returns an <code>XMLHttpRequest</code> object, which provides functionality for Ajax requests.

Table 12-2 window Methods

Method	Description
<code>addEventListener()</code>	Registers an event handler for the <code>Window</code> object.
<code>alert()</code>	Displays an alert box.
<code>atob()</code>	Decodes base64 data.
<code>attachEvent()</code>	Registers an event handler on the <code>Window</code> object.
<code>back()</code>	Moves back one page in the user's history.
<code>blur()</code>	Removes the window from focus.
<code>btoa()</code>	Encodes a string in base64.

Method	Description
<code>clearInterval()</code>	Stops a currently running timer set up through <code>setInterval()</code> .
<code>clearTimeout()</code>	Stops a currently running timer set up through <code>setTimeout()</code> .
<code>close()</code>	Closes the window.
<code>confirm()</code>	Displays a confirmation dialog box.
<code>createPopup()</code>	Creates a pop-up window.
<code>detachEvent()</code>	Removes an event handler that was created using <code>attachEvent()</code> .
<code>dispatchEvent()</code>	Sends an event on the <code>Window</code> object.
<code>escape()</code>	Encodes a string by replacing some characters with their hex equivalent. Use <code>unescape()</code> to revert.
<code>execScript()</code>	Executes the script in the given language.
<code>find()</code>	Searches for the text in the document and highlights it if it is found.
<code>focus()</code>	Gives the window the focus.
<code>forward()</code>	Moves one page forward in the user's history.
<code>getComputedStyle()</code>	Gets the computed style for the given object.

<code>getSelection()</code>	Returns a <code>selectionRange</code> object that provides data on the selected region of the page.
<code>home()</code>	Loads the user's home page.
<code>matchMedia()</code>	Returns a <code>MediaQueryList</code> object representing the specified media query string.
<code>moveBy()</code>	Moves the position of the window by the given x, y values.
<code>moveTo()</code>	Moves the position of the window to the given x, y values.
<code>mozRequestAnimationFrame()</code>	Alerts the browser that an animation is in progress and a repaint should be scheduled.
<code>navigate()</code>	Loads the given URL.
<code>open()</code>	Opens a new window.
<code>openDialog()</code>	Opens a new dialog window.
<code>postMessage()</code>	Sends a message from one window to another.
<code>print()</code>	Calls the browser's print dialog.
<code>prompt()</code>	Displays a prompt dialog.
<code>removeEventListener()</code>	Removes an event handler that was created with <code>addEventListener()</code> .
<code>resizeBy()</code>	Resizes the window by the given x,y values.

<code>resizeTo()</code>	Resizes the window to the given x,y values.
<code>scroll()</code>	Scrolls the document to the given x,y values.
<code>scrollBy()</code>	Scrolls the document by the given x,y values.
<code>scrollByLines()</code>	Scrolls the document by the given number of lines.
<code>scrollByPages()</code>	Scrolls the document by the given number of pages.
<code>scrollTo()</code>	Scrolls the document to the given x,y position.
<code>setCursor()</code>	Changes the cursor for the window.
<code>setInterval()</code>	Creates a timer that calls a function each time a number of milliseconds have passed.
<code>setTimeout()</code>	Creates a timer that calls a function once a number of milliseconds have passed.
<code>showModalDialog()</code>	Displays a modal dialog box.
<code>showModelessDialog()</code>	Displays a modeless dialog box.
<code>sizeToContent()</code>	Sizes the window based on its content.
<code>stop()</code>	Stops the loading of the window.
<code>toStaticHTML()</code>	Removes dynamic HTML from an HTML fragment.
<code>unescape()</code>	Decodes a string that has been encoded in hexadecimal format.

Given the wide range of properties and methods in the Window object, we focus in this chapter on the creation and management of windows. Subsequent chapters will address some of the important objects such as Navigator, Document, Screen, and so on.

Dialogs

We begin our discussion of the application of the Window object by presenting how to create three types of special windows known generically as dialogs. A *dialog box*, or simply *dialog*, is a small window in a graphical user interface that “pop ups,” requesting some action from a user. The three types of basic dialogs supported directly by JavaScript include alerts, confirms, and prompts. How these dialogs are natively implemented is somewhat rudimentary, but in the next section we’ll see that once we can create our own windows we can replace these windows with our own.

alert()

The Window object’s `alert()` method creates a special small window with a short string message and an OK button, as shown here:



NOTE The visual representation of an alert dialog can vary widely between browsers and may or may not include an icon and information about what is issuing the alert. The icon may be browser focused or unfortunately may look like a warning, regardless of the meaning of the message being presented.

The basic syntax for alert is

```
window.alert(string);
```

or, for shorthand, we just use

```
alert(string);
```

as the Window object can be assumed. The string passed to any dialog such as an alert may be either a variable or the result of an expression. If you pass another form of data, it will be coerced into a string. All of the following examples are valid uses of the alert method:

```
alert("Hi there from JavaScript!");  
alert("Hi "+username+" from JavaScript");  
var messageString = "Hi again!";  
alert(messageString);
```

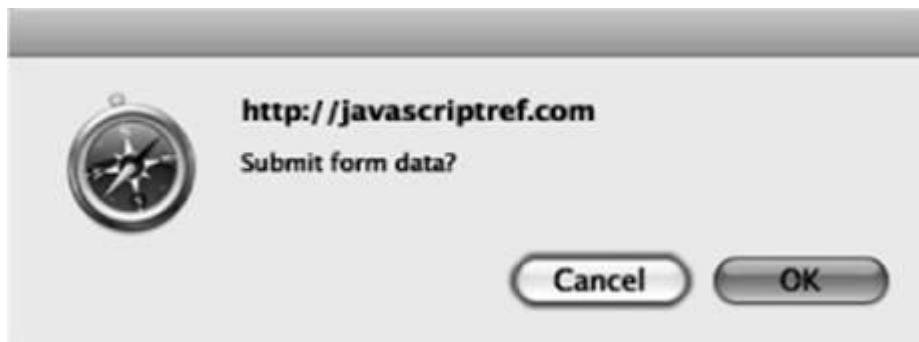
An alert window is *page modal*, meaning that it must receive focus and be cleared before the user is allowed to continue activity with the page.

One common use of alert dialogs is debugging messages. While this seems an acceptable use of alerts, it is generally more appropriate to

pipe such messages to a browser's console using the `console.log()` method. Not only does this keep the message outside the view of the casual user, but often you need to issue many debugging traces, and the alert's modal nature may be both annoying and inappropriate, depending on what the code is doing.

`confirm()`

The `confirm()` method creates a window that displays a message for a user to respond to by pressing either an OK button to agree with the message or a Cancel button to disagree with the message. A typical rendering is shown here:



The writing of the confirmation question may influence the usability of the dialog significantly. Many confirmation messages are best answered with a Yes or No button, rather than an OK or a Cancel button, as shown by the following dialog:



Unfortunately, using the basic JavaScript confirmation method, there is no possibility of changing the button strings, so choose your message wisely. Fortunately, later we'll see it is quite possible to write your own form of confirmation.

The basic syntax of the `confirm()` method is

```
window.confirm(string);
```

or simply

```
confirm(string);
```

where *string* is any valid string variable, literal, or expression that either evaluates to or will be coerced into a string that will be used as the confirmation question.

The `confirm()` method returns a Boolean value, which indicates whether or not the information was confirmed, true if the OK button was pressed, and false if the Cancel button was pressed or the dialog was closed, as some older browsers allow for. The return value can be saved, like so:

```
answer = confirm("Do you want to do this?");
```

or the method call itself can be used within any construct that uses a Boolean expression such as an if statement:

```
if (confirm("Do you want ketchup on that?")) {  
    alert("Pour it on!");  
}  
else {  
    alert("Hold the ketchup.");  
}
```

Like the `alert()` method, confirmation dialogs should be browser modal.

The following example shows how the `alert` and `confirm` can be used:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JavaScript Power - alert() and confirm()</title>
<script>

window.onload = function () {

    document.getElementById("destructBtn").onclick = function () {
        if (confirm("Are you sure you want to destroy this page?"))
            alert("What? You thought I'd actually let you do that!?");
        else
            alert("That was close!");
    };
};
</script>
</head>
<body>
<h1>The Mighty Power of JavaScript!</h1>
<form>
  <input type="button" id="destructBtn" value="Destroy this page!">
</form>
</body>
</html>
```

ONLINE <http://javascriptref.com/3ed/ch12/alertconfirm.html>

prompt()

A prompt window invoked by the `prompt()` method of the Window object is a small data collection dialog that prompts the user to enter a short line of data, as shown here:



The `prompt()` method takes two arguments. The basic syntax is shown here:


```
resultvalue = window.prompt(prompt string, default value string);
```

The first parameter is a string that displays the prompt value, and the second is a default value to put in the prompt window. The method returns a string value that contains the value entered by the user in the prompt.

The shorthand `prompt()` is almost always used instead of `window.prompt()`, and occasionally programmers will accidentally use only a single value in the method:

```
var result = prompt("What is your least favorite coding mistake?");
```

However, in many browsers you may see that a value of `undefined` is placed in the prompt line. You should set the second parameter to an empty string to keep this from happening:

```
var result = prompt("What is your least favorite coding mistake?","");
```

When using the `prompt()` method, it is important to understand what is returned. If the user presses the Cancel button in the dialog or the close box, a value of `null` will be returned. It is always a good idea to check for this. Otherwise, a string value will be returned. Programmers should be careful to convert prompt values to the appropriate type using `parseInt()`, `parseFloat()`, or another type conversion scheme if they do not want a string value.

The following example shows the `prompt()` method in action:


```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>prompt()</title>
</head>
<body>
<h1>JavaScript Guru 1.0</h1>
<hr>
<form>
  <input type="button" id="guruBtn" value="Ask the Guru">
</form>
<script>
window.onload = function () {
  document.getElementById("guruBtn").onclick = function () {
    var question = prompt("What is your question o' seeker of JS knowledge?","");
    if (question) {
      alert("Good question. Who knows?");
    }
    else {
      alert("At least you could ask a question.");
    }
  };
};
</script>
</body>
</html>
```

ONLINE <http://javascriptref.com/3ed/ch12/prompt.html>

Emerging and Proprietary Dialog Methods

The format of these last three dialogs leaves a little to be desired. We explore a few emerging and proprietary mechanisms before moving on to creating our own windows from scratch.

showModalDialog()

Internet Explorer introduced a modal window, which was later incorporated into the HTML5 standard. Like a standard dialog, this more generic window is modal to the page and must be dismissed before moving on. The basic syntax for creating a modal dialog is

```
window.showModalDialog(URL of dialog, arguments, features);
```

where

- *URL of dialog* is a URL of the document to display.
- *arguments* are any objects or values you wish to pass to the modal dialog.
- *features* is a semicolon-separated list of display features for the dialog.

The *features* string should be the same as what is supported by `window.open()`, which will be covered shortly, though MSDN syntax shows some variations that will likely change as this is standardized.

A simple example of this method is shown here:

```
window.showModalDialog("customdialog.html", window,  
"dialogHeight: 150px; dialogWidth: 300px; center: Yes;  
help: No; resizable: No; status: No;");
```

The `showModalDialog()` method also returns a value. To accomplish this, set the `window.returnValue` property in the dialog and the return of this value will happen automatically. This mechanism allows for the simple creation of user prompt and confirmation dialogs, which must return a value.

The second parameter can be an arbitrary argument. This object is accessible within the dialog with the `window.dialogArguments` property. Internet Explorer supports some additional dialog properties, including `window.dialogWidth`, `window.`

`dialogHeight`, `window.dialogTop`, and `window.dialogLeft`.

showModelessDialog()

Microsoft also introduced a modeless window that is very different from a modal dialog. While both dialog boxes always maintain focus, a modeless window does allow you to focus in the window that created the dialog. A common use for this might be to display help or other very contextual useful information. However, while different in function, a modeless window is syntactically similar to the modal dialog.

```
windowreference = window.showModelessDialog(URL of dialog, arguments, features)
```

The method parameters are the same, but the returned value is not a value created within the dialog. Instead, it's a reference to the created window in case it is manipulated at a later time. This would be similar, then, to the value returned by `window.open()`. A simple example of the syntax for creating a modeless window is shown here:

```
var myWindow = window.showModelessDialog("customdialog.html", window,  
"dialogHeight: 150px; dialogWidth: 300px; center: Yes; help: No;  
resizable: No; status: No;");
```

Readers should remember that this syntax is not currently covered under the HTML5 specification.

createPopup()

A special window form supported by Microsoft is a generic form of pop-up window. Creating a pop-up is very simple—just use the `window.createPopup()`, which takes no arguments and returns a handle to the newly created window:

```
var myPopup = window.createPopup();
```

These windows are initially created but are hidden. They are later revealed using the popup object's `show()` method and hidden using `hide()`, as shown here:

```
myPopup.show(); // displays created pop-up  
myPopup.hide(); // hides the pop-up
```

The value of Microsoft's special pop-ups may not be obvious until you consider that they have more control over their appearance than standard JavaScript dialogs. In fact, initially you could even remove the chrome of the displayed window. Chromeless windows were, however, abused by those looking to phish end-user passwords. Often these windows were used to position over the URL bar or perform other trickery. Today their use is severely limited in Internet Explorer.

A complete example showing the use of all three of these unusual dialog windows is shown here:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Special Dialog Windows</title>
<script>
var myPopup = null;
function showPopup() {
    if (!myPopup) {
        myPopup = window.createPopup();
    }
    var popupBody = myPopup.document.body;
    popupBody.style.backgroundColor = "#ffff99";
    popupBody.style.border = "solid black 1px";
    popupBody.innerHTML = "Click outside this window to close, or press hide
button.";
    myPopup.show(50, 100, 350, 25, document.body);
}
function hidePopup() {
    myPopup.hide();
}
function makeModalDialog() {
    // modal.html has the modal dialog information in it
    showModalDialog("modal.html", window,
        "status:false;dialogWidth:300px;dialogHeight:100px;help:no;status:no;");
}

function makeModelessDialog() {
    var myModelessDialog =
showModelessDialog("", window,
"status:false;dialogWidth:200px;dialogHeight:300px;help:no;status:no;");
    modelessBody = myModelessDialog.document.body;
    modelessBody.style.backgroundColor = "#ffcc33"

    var HTMLoutput = "<html><head><title>Modeless Dialog</title></head>";
    HTMLoutput += "<body><h1>Important messages in this modeless window</h1><hr>";
    HTMLoutput += "dialogLeft: " + myModelessDialog.dialogLeft + "<br>";
    HTMLoutput += "dialogTop: " + myModelessDialog.dialogTop + "<br>";
    HTMLoutput += "dialogWidth: " + myModelessDialog.dialogWidth + "<br>";
    HTMLoutput += "dialogHeight: " + myModelessDialog.dialogHeight + "<br>";
    HTMLoutput += "<form><div style='align:center;'>
        <input type='button' value='close' onclick='self.close();'>";
    HTMLoutput += "</div></form></body></html>";

    modelessBody.innerHTML = HTMLoutput;
}

```

```
window.onload = function() {
  document.getElementById("modalButton").onclick = makeModalDialog;
  document.getElementById("modelessButton").onclick = makeModelessDialog;
  document.getElementById("showButton").onclick = showPopup;
  document.getElementById("hideButton").onclick = hidePopup;
};
</script>
</head>
<body>
<form>
<input type="button" value="Modal Dialog" id="modalButton">
<input type="button" value="Modeless Dialog" id="modelessButton">
<input type="button" value="Show Popup" id="showButton">
<input type="button" value="Hide Popup" id="hideButton">
</form>
</body>
</html>
```

ONLINE <http://javascriptref.com/3ed/ch12/specialdialogs.html>

Opening and Closing Generic Windows

While the `alert()`, `confirm()`, and `prompt()` methods create specialized windows quickly, it is often desirable to open arbitrary windows to show a Web page or the result of some calculation. The Window object methods `open()` and `close()` are used to create and destroy a Window, respectively.

When you open a Window, you can set its URL, name, size, buttons, and other attributes, such as whether or not the window can be resized. The basic syntax of this method is

```
window.open(URL, name, features, replace)
```

where

- *URL* is a URL that indicates the document to load into the window.
- *name* is the name for the window (which is useful for referencing it later on using the **target** attribute of HTML links).
- *features* is a comma-delimited string that lists the features of the window.

- *replace* is an optional Boolean value (true or false) that indicates whether or not the URL specified should replace the window's contents. This would apply to a window that was already created.

A simple example of this method is

```
var secondwindow = window.open("http://www.google.com", "google", "height=300, width=600, scrollbars=yes");
```

This would open a window to Google with a height of 300 pixels, a width of 600 pixels, and scroll bars, as shown here:



Of course, because we spawn the window directly, your browser's pop-up blocker might get in the way. We'll discuss handling that in a bit, but for now we might want to focus on triggering window creation rather than just having it occur directly. Obviously, there are a variety of ways programmers might trigger the creation of windows, but most often links or buttons are used, as shown here:

```
<a href="#" id="openLink">Open Window</a>

<form>
  <input type="button" id="openBtn" value="Open Window">
</form>

<script>
function openWindow() {
  // note global being used here
  secondWindow = open("http://www.google.com", "google", "height=300,
width=600,scrollbars=yes");
}

window.onload = function() {
  document.getElementById("openLink").onclick = openWindow;
  document.getElementById("openBtn").onclick = openWindow;
};
</script>
```

ONLINE <http://www.javascriptref.com/3ed/ch12/windowopentriigger.html>

One useful feature from the dialogs in the previous section was the ability to send arguments to the new dialog box. This is not possible with `window.open()`, as `window.open()` does not receive an arguments parameter. However, `window.openDialog()` functions mostly the same as `window.open()`, with the exception that it has the ability to send arguments. The arguments would come after the features parameter, and there is no limit to the number of arguments that can be passed. The arguments can be accessed within the new window with the `window.arguments` property.

Once a window is open, the `close()` method can be used to close it. For example, the following fragment presents buttons to open and close a window. Make sure to notice the use of the `secondWindow` variable that contains the instance of the Window object created:

```

<form>
  <input type="button" id="openBtn" value="Open Window">
  <input type="button" id="closeBtn" value="Close Window">
</form>
<script>
function openWindow() {
  // note global being used here
  secondWindow = open("http://www.google.com", "google", "height=300,width
=600,scrollbars=yes");
}

function closeWindow() {
  secondWindow.close();
}

window.onload = function() {
  document.getElementById("openBtn").onclick = openWindow;
  document.getElementById("closeBtn").onclick = closeWindow;
};
</script>

```

First, we notice that the variable *secondWindow* is in the global scope. Certainly not the best way to code, but we must have a reference to the created window in order to run the `close()` method. Obviously, a better solution would be to use some wrapper object to house any global references we need, like so:

```

var JSREF = {};
JSREF.secondWindow = open("http://www.google.com", "example",
                          "height=300,width=600,scrollbars=yes");

```

Next, we should address the usage of the `close()` method, which is rather dangerous. If the new window does not yet exist, the script will throw an error. Reload the previous example and press the Close button immediately, and you should get an error. In order to safely close a Window, you first need to look for the object and then try to close it. Consider the following if statement that looks to see if the *JSREF.secondWindow* value is defined before looking at it; then it will look at the `closed` property to make sure it is not already closed:

```

if (JSREF.secondWindow && !JSREF.secondWindow.closed)
  JSREF.secondWindow.close();

```

Notice that this previous example actually specifically relies on short-circuit evaluation, because if the value in *JSREF.secondWindow* is undefined, then trying to look at its `closed` property would throw an

error. The following short example shows the safe use of the Window methods and properties discussed so far:

```
<form>
  <input type="button" value="Open Window" id="openBtn">
  <input type="button" value="Close Window" id="closeBtn">
  <input type="button" value="Check Status" id="statusBtn">
</form>
<script>
var JSREF = {};
window.onload = function() {

    document.getElementById("openBtn").onclick = function () {
        JSREF.secondWin= open("http://www.google.com", "example",
            "height=300,width=600,scrollbars=yes");
    };

    document.getElementById("closeBtn").onclick = function() {
        if (JSREF.secondWin)
            JSREF.secondWin.close();
    };

    document.getElementById("statusBtn").onclick = function() {
        if (JSREF.secondWin)
            alert(JSREF.closed);
        else
            alert("JSREF.secondWin undefined");
    };
};
</script>
```

ONLINE <http://www.javascriptref.com/3ed/ch12/windowclose.html>

TIP If you create a window within an HTML tag's event handler attribute, remember that the variable scope will not be known outside of that tag. If you want to control a window, very likely you will need to define it in the global scope.

Besides checking for the existence of windows before closing, be aware that you cannot close windows that you have not created, particularly if security privileges have not been granted to the script.

In fact, you may even have a hard time closing the main browser window. For example, if you have a statement such as `window.close()` in the main browser window running the script, it may be denied—some older browsers may prompt you to confirm and others will close the window down without warning.

Window Features

When creating new windows within `window.open()`, there are a number of possibilities for the *feature* parameter, which is quite rich and allows you to set the height, width, scroll bars, and a variety of other window characteristics. The possible values for this parameter are detailed in *Table 12-3*.

Table 12-3 Feature Parameter Values for `window.open()`

Feature Parameter	Value	Description	Example
<code>alwaysLowered</code>	yes/no	Indicates whether or not a window should always be lowered under all other windows. It does have a security risk and may be restricted in browsers.	<code>alwaysLowered=no</code>
<code>alwaysRaised</code>	yes/no	Indicates whether or not the window should always stay on top of other windows. Similar to lowering, some browsers may limit this value.	<code>alwaysRaised=no</code>
<code>centerscreen</code>	yes/no	Indicates whether or not the window should be centered in relation to its parent's size and position. Requires parameter <code>chrome=yes</code> .	<code>centerscreen=yes</code>
<code>chrome</code>	yes/no	Indicates whether or not the page should be the only content without any of the browser's interface elements. Supported by Firefox only and requires <code>UniversalBrowserWrite</code> privilege.	<code>chrome=true</code>
<code>close</code>	yes/no	Indicates whether or not the Close button is displayed in dialog windows. Supported by Firefox only and requires <code>UniversalBrowserWrite</code> privilege.	<code>close=no</code>

dialog	yes/no	Indicates whether or not all icons (restore, minimize, maximize) should be removed from the window's titlebar, leaving only the close button.	dialog=yes
dependent	yes/no	Indicates whether or not the spawned window is truly dependent on the parent window. Dependent windows are closed when their parents are closed, while others stay around.	dependent=yes
directories	yes/no	Indicates whether or not the directories button on the browser window should show. This parameter is obsolete and is replaced by <code>personalbar</code> .	directories=yes
fullscreen	yes/no	Specifies whether or not the window should take over the full screen. Supported by Internet Explorer only.	fullscreen=yes
height	pixel value	Sets the height of the window including any browser chrome.	height=100
hotkeys	yes/no	Indicates whether or not the hot keys for the browser-beyond-browser essential hot keys such as quit should be disabled in the new window.	hotkeys=no

innerHeight	pixel value	Sets the height of the inner part of the window where the document shows.	innerHeight=200
innerWidth	pixel value	Sets the width of the inner part of the window where the document shows.	innerWidth=300
left	pixel value	Specifies where, relative to the screen origin, to place the window.	left=10
location	yes/no	Specifies whether or not the location bar should show on the window.	location=no
menubar	yes/no	Specifies whether or not the menu bar should be shown.	menubar=yes
minimizable	yes/no	Indicates whether or not the minimize icon should be shown in the case that <code>dialog=yes</code> .	minimizable=yes
modal	yes/no	Indicates whether or not the window should display in modal mode. Supported by Firefox only and requires <code>UniversalBrowserWrite</code> privilege.	modal=yes
outerHeight	pixel value	Sets the height of the outer part of the window, including the chrome.	outerHeight=300
outerWidth	pixel value	Sets the width of the outer part of the window, including the chrome.	outerWidth=300

<code>personalbar</code>	yes/no	Indicates whether or not the Links or Bookmarks Bar should show. Replaces the parameter <code>directories</code> .	<code>personalbar=true</code>
<code>resizable</code>	yes/no	Indicates whether or not the user should be able to resize the window.	<code>resizable=no</code>
<code>screenx</code>	pixel value	Deprecated Netscape syntax that indicates the distance to the left in pixels from the screen origin where the window should be opened. Use <code>left</code> instead.	<code>screenx=100</code>
<code>screeny</code>	pixel value	Deprecated Netscape syntax that indicates the distance up and down from the screen origin where the window should be opened. Use <code>top</code> instead.	<code>screeny=300</code>
<code>scrollbars</code>	yes/no	Indicates whether or not scroll bars should show.	<code>scrollbars=no</code>
<code>status</code>	yes/no	Indicates whether or not the status bar should show.	<code>status=no</code>
<code>titlebar</code>	yes/no	Indicates whether or not the title bar should show.	<code>titlebar=yes</code>

<code>toolbar</code>	yes/no	Indicates whether or not the toolbar menu should be visible.	<code>toolbar=yes</code>
<code>top</code>	pixel value	Indicates the position down from the top corner of the screen to position the window.	<code>top=20</code>
<code>width</code>	pixel value	Sets the width of the window, including outside browser; thus, you may want to use <code>innerWidth</code> instead. However, note that <code>innerWidth</code> was not supported in Internet Explorer before version 8.	<code>width=300</code>
<code>z-lock</code>	yes/no	Specifies whether or not the z-index should be set so that a window cannot change its stacking order relative to other windows even if it gains focus.	<code>z-lock=yes</code>

NOTE Typically, in modern JavaScript implementations, you can use 1 for yes and 0 for no for the features using yes/no values. However, for safety and backward compatibility, the yes/no syntax is preferred.

Oftentimes when using the `open()` method, you may want to create strings to hold the options rather than use a string literal. However, when the features are specified, remember that they should be set one at a time with comma separators and no extra spaces. For example,

```
var windowOptions = "directories=no,location=no,width=300,height=300";  
var spawnedWindow = open("http://www.google.com", "googWin", windowOptions);
```

The next example is useful for experimenting with all the various window features that can be set. It also will display the JavaScript string required to create a particular window in a text area so it can be used in a script.

NOTE In addition to the height and width properties, in Firefox it is possible to set the size of the window through the `window.sizeToContent()` method, which will set the size of the window to fit the content rendered.

```
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="utf-8">  
<title>window.open()</title>  
<style>  
  fieldset.checkbox label {width: 150px; display: block;}  
  fieldset.checkbox input {float: right;}  
</style>  
</head>  
<body>  
<form>  
<fieldset>  
  <legend>Window Basics</legend>  
  <label>URL:  
    <input type="text" id="windowurl" size="30" maxlength="300"  
      value="http://www.google.com">  
  </label>  
  <br>  
  <label>Window Name:  
    <input type="text" id="windowname" size="30"  
      maxlength="300" value="secondwindow">  
  </label>  
  <br>  
</fieldset>
```

```

<fieldset>
  <legend>Size</legend>
  <select id="hwoptions">
    <option selected>Height/Width</option>
    <option>innerHeight/innerWidth</option>
    <option>outerHeight/outerWidth</option>
  </select>

  <br><br>
  <div id="hw">
    <label>Height:<input type="text" id="height" size="4"
maxlength="4" value="100"></label>
    <label>Width:<input type="text" id="width" size="4"
maxlength="4" value="100"></label>
  </div>
  <div id="ihw" style="display:none;">
    <label>innerHeight:<input type="text" id="innerHeight" size="4"
maxlength="4" value="100"></label>
    <label>innerWidth:<input type="text" id="innerWidth" size="4"
maxlength="4" value="100"></label>
  </div>
  <div id="ohw" style="display:none;">
    <label>outerHeight:<input type="text" id="outerHeight" size="4"
maxlength="4" value="100"></label>
    <label>outerWidth:<input type="text" id="outerWidth" size="4"
maxlength="4" value="100"></label>
  </div>
</fieldset>

<fieldset>
  <legend>Position</legend>
  <label>Top:<input type="text" id="top" size="4" maxlength="4" value="100"></label>
  <label>Left:<input type="text" id="left" size="4" maxlength="4" value="100"></label>
</fieldset>

<fieldset class="checkbox">
  <legend>Display Features</legend>

  <label>alwaysLowered: <input type="checkbox" id="alwaysLowered"></label>
  <label>alwaysRaised: <input type="checkbox" id="alwaysRaised"></label>
  <label>dependent: <input type="checkbox" id="dependent"></label>
  <label>dialog: <input type="checkbox" id="dialog"></label>
  <label>directories: <input type="checkbox" id="directories"></label>
  <label>fullscreen: <input type="checkbox" id="fullscreen"></label>
  <label>hotkeys: <input type="checkbox" id="hotkeys"></label>
  <label>location: <input type="checkbox" id="location"></label>
  <label>menubar: <input type="checkbox" id="menubar"></label>
  <label>minimizable: <input type="checkbox" id="minimizable"></label>
  <label>personalbar: <input type="checkbox" id="personalbar"></label>
  <label>resizable: <input type="checkbox" id="resizable"></label>
  <label>scrollbars: <input type="checkbox" id="scrollbars"></label>
  <label>status: <input type="checkbox" id="status"></label>

```

```

<label>titlebar: <input type="checkbox" id="titlebar"></label>
<label>toolbar: <input type="checkbox" id="toolbar"></label>
<label>z-lock: <input type="checkbox" id="z-lock"></label>
</fieldset>
<br>
<input type="button" value="Create Window" id="openButton">
<input type="button" value="Close Window" id="closeButton">
<hr>
<h2>JavaScript window.open() Statement</h2>
<textarea id="jrcode" rows="4" cols="80"></textarea>
</form>
<script>
function createFeatureString() {
  var featurestring = "";
  var elements = document.getElementsByTagName("input");
  var numelements = elements.length;
  for (var i = 0; i < numelements; i++)
    if (elements[i].type == "checkbox") {
      if (elements[i].checked) {
        featurestring += elements[i].id+"=yes,";
      }
      else {
        featurestring += elements[i].id+"=no,";
      }
    }
}

var selection = document.getElementById("hwoptions").selectedIndex;
if (selection == 0) {
  featurestring += "height="+document.getElementById("height").value+",";
  featurestring += "width="+document.getElementById("width").value+",";
}
else if (selection == 1) {
  featurestring += "innerHeight="+document.getElementById("innerHeight").value+",";
  featurestring += "innerWidth="+document.getElementById("innerWidth").value+",";
}
else if (selection == 2) {
  featurestring += "outerHeight="+document.getElementById("outerHeight").value+",";
  featurestring += "outerWidth="+document.getElementById("outerWidth").value+",";
}

featurestring += "top="+document.getElementById("top").value+",";
featurestring += "left="+document.getElementById("left").value;
return featurestring;
}

function openWindow() {
  var features = createFeatureString();
  var url = document.getElementById("windowurl").value;
  var name = document.getElementById("windowname").value;
  theNewWindow = window.open(url,name,features);
  if (theNewWindow)
    document.getElementById("jrcode").value =
      "window.open('+url+', '"+name+', '"+features+'');"
  else
    document.getElementById("jrcode").value = "Error: JavaScript Code Invalid";
}

function closeWindow() {
  if (window.theNewWindow)
    theNewWindow.close();
}

```



```
}  
  
function updateHW() {  
    var hw = document.getElementById("hw");  
    var ihw = document.getElementById("ihw");  
    var ohw = document.getElementById("ohw");  
    var selection = document.getElementById("hwoptions").selectedIndex;  
  
    hw.style.display = "none";  
    ihw.style.display = "none";  
    ohw.style.display = "none";  
  
    if (selection == 0) {  
        hw.style.display = "";  
    }  
    else if (selection == 1) {  
        ihw.style.display = "";  
    }  
    else if (selection == 2) {  
        ohw.style.display = "";  
    }  
}  
  
window.onload = function() {  
    document.getElementById("openButton").onclick = openWindow;  
    document.getElementById("closeButton").onclick = closeWindow;  
    document.getElementById("hwoptions").onchange = updateHW;  
};  
</script>  
</body>  
</html>
```

ONLINE <http://javascriptref.com/3ed/ch12/windowopen.html>

A rendering of the previous example is shown in *Figure 12-1*.

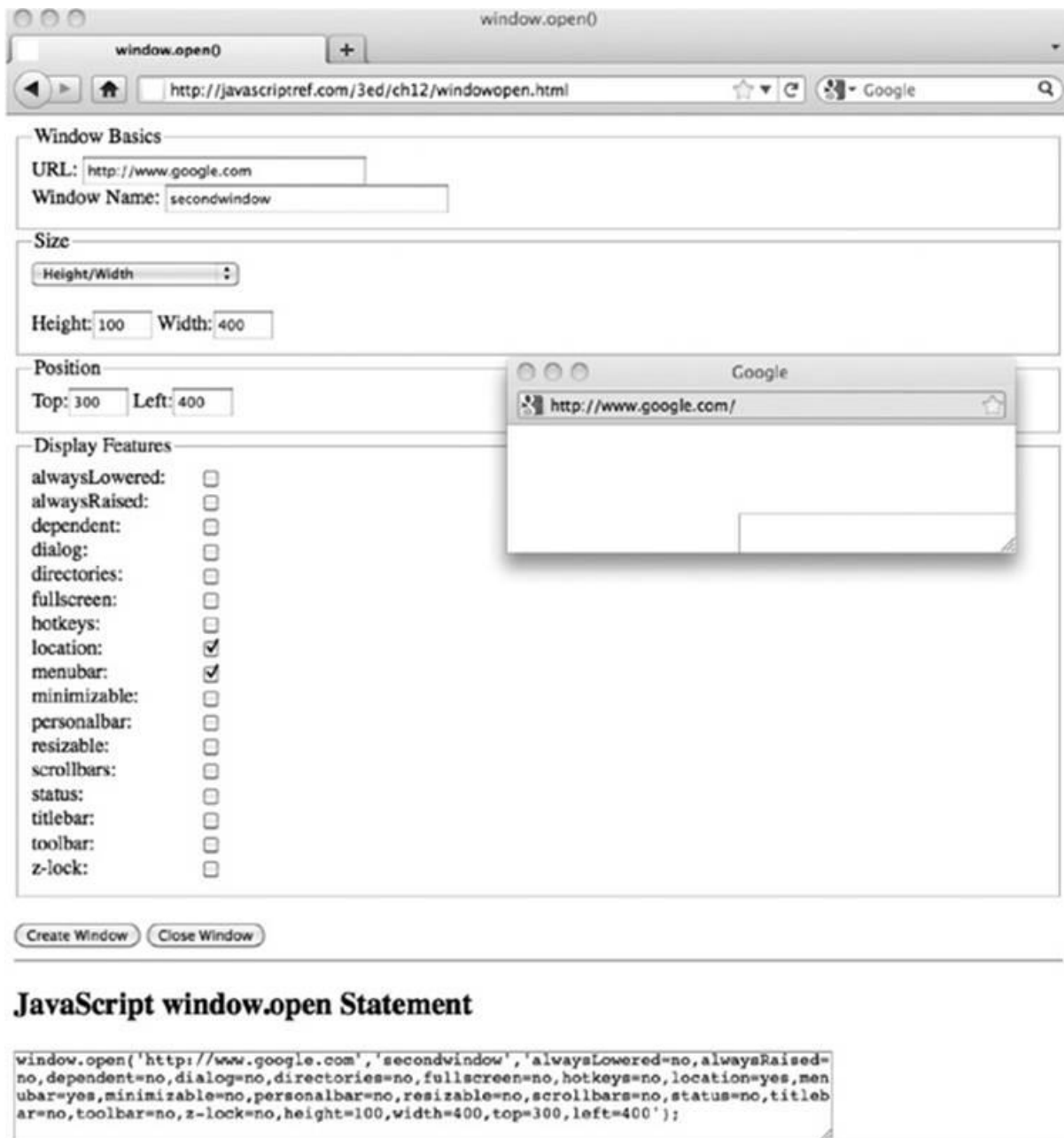


Figure 12-1 Rendering of window.open() example

Sadly, if you try the previous example, you will find that many features no longer work. For example, the stacking properties, such as alwaysLowered, alwaysRaised, and z-lock, likely will not work due to security abuses. You will find also that it is impossible to size windows smaller than say 50 × 50 pixels for similar reasons. You can't even hide the location, as it will always show up in most browsers. Modifying the menu bars is just as problematic. We'll see that, given these spotty issues, custom windows are fading into the

past in Web development. Some changes to the HTML5 specification, though, do give us hope that the situation may be rectified someday.

Detecting and Controlling window Chrome

HTML5 introduces a standard way to understand what chrome elements such as menus may be showing on the main window or any spawned window. A number of BarProp objects exist, including the following:

```
window.locationbar  
window.menubar  
window.personalbar  
window.scrollbars  
window.statusbar  
window.toolbar
```

Each of these objects corresponds to the similarly named browser menu. The object currently contains a single property called `visible` that indicates whether or not the menu is showing. When a script is executing in a privileged mode, it may be possible to set this value for control, but at this point it certainly is not allowed outside of that. A simple example of this emerging aspect of Window is shown here, and readers are encouraged to explore these objects further, as they are likely to be expanded on:

```
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="utf-8">  
<title>BarProp Objects</title>  
<script>  
function showBarProps() {  
  var str = "";  
  str += "window.locationbar.visible: " + window.locationbar.visible + "<br>";  
  str += "window.menubar.visible: " + window.menubar.visible + "<br>";  
  str += "window.personalbar.visible: " + window.personalbar.visible + "<br>";  
  str += "window.scrollbars.visible: " + window.scrollbars.visible + "<br>";  
  str += "window.statusbar.visible: " + window.statusbar.visible + "<br>";  
  str += "window.toolbar.visible: " + window.toolbar.visible + "<br>";  
  document.getElementById("message").innerHTML = str;  
}
```

```
function hideMenuBar() {
    window.menuBar.visible = false;
}

window.onload = function () {
    document.getElementById("showBtn").onclick = showBarProps;
    document.getElementById("hideBtn").onclick = hideMenuBar;
};
</script>
</head>
<body>
<form>
  <input type="button" value="Show Bar Props" id="showBtn">
  <input type="button" value="Hide Menu Bar" id="hideBtn">
  <br>
  <div id="message"></div>
</form>
</body>
</html>
```

ONLINE <http://javascriptref.com/3ed/ch12/barprop.html>

Practicalities of Spawned Windows

The reality of windows in browsers is less than clean. Adding content to newly created windows can be a chore, depending on how it is done, and sloppy mistakes can cause trouble. Security concerns abound for the size and position of windows, and pop-up blockers can cause lots of trouble. In fact, for many JavaScript developers, subwindows have long since been abandoned for pseudo-windows in the form of `<div>`-based overlays.

Building window Contents

When opening a new window, it is quite easy to populate it if you have an existing document at some URL:

```
var myWindow = open("somefile.html", "mywin", "height=400, width=600");
```

However, if you need to create a window directly, there are a number of ways it can be done. First, you might fall to traditional `document.write()` use, like so:

```

var myWindow = open("", "mywin", "height=400,width=600");

myWindow.document.writeln("<!DOCTYPE html>");
myWindow.document.writeln("<html>");
myWindow.document.writeln("<head>");
myWindow.document.writeln("<meta charset='utf-8'>");
myWindow.document.writeln("<title>New Window</title>");
myWindow.document.writeln("</head>");
myWindow.document.writeln("<body>");
myWindow.document.writeln("<h1>Hi from the new window!</h1>");
myWindow.document.writeln("</body>");
myWindow.document.writeln("</html>");

// make sure to close the document
myWindow.document.close();
myWindow.focus();

```

Notice a few points here. First, the use of `document.close()`. Without this statement, some browsers will assume there is more content coming, so the window will appear never to finish loading, while others will close implicitly. Given this variability, it is quite important to have this method call. Second, notice the use of `document.writeln()`. If we are looking to build clean-looking HTML source, this inserts newlines. However, unless we expect people to view the source of a spawned window with a debugging tool, this is kind of pointless. We certainly could use `document.write()` and add in `\n` characters if we wish:

```

myWindow.document.write("<!DOCTYPE html>\n<html>\n<head>\n");

```

Of course, even having a multitude of `document.write()` statements seems wrong, and indeed it is, as it may cause performance concerns. Instead, we may gather content for output in a string variable and output it at once:

```

var myWindow = open("", "mywin", "height=400,width=600");
var str = "";
str += "<!DOCTYPE html><html><head><meta charset='utf-8'>";
str += "<title>New Window</title></head><body>";
str += "<h1>Hi from the new window!</h1><body></html>";

// write the entire string at once
myWindow.document.write(str);

// make sure to close the document
myWindow.document.close();
myWindow.focus();

```

You may opt to perform DOM calls to create the elements, but this seems to us to be an exercise in lots of code for little value. If you must go that route, you could instead employ the `innerHTML` property, like so:

```

var myWindow = open("", "mywin", "height=400,width=600");
myWindow.document.title = "New Window";
myWindow.document.body.innerHTML = "<h1>Hi from the new window!</h1>";

// make sure to close the document
myWindow.document.close();
myWindow.focus();

```

Notice that we don't construct all the pieces of the document. We certainly could have, but the browser will scaffold a base tree for us anyway.

Reading and Writing Existing Windows

We do not use `document.write()` after a window is created unless we desire to wipe the entire page clean. Instead, we rely on DOM methods to insert and change the HTML in the new document at will. The only difference is that now you must make sure to use the new window's name when accessing a DOM method or property. For example, if you had a window called *newWindow*, you would use statements such as the following to retrieve a particular element in the other window:

```

var currentElement = newWindow.document.getElementById("myheading");

```

The following simple example shows how information entered in one window can be used to create an element in another window:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>DOM Window Add</title>
<script>
var myWindow;
function domWindowAdd() {
  if ((window.myWindow) && (myWindow.closed == false)) {
    var str = document.getElementById("textToAdd").value;
    var theString = myWindow.document.createTextNode(str);
    var theElement = myWindow.document.createElement("h1");
    theElement.appendChild(theString);
    myWindow.document.body.appendChild(theElement);
    myWindow.focus();
  }
}

function createWindow() {
  myWindow = open("", "mywin", "height=300,width=300");
  myWindow.document.writeln("<!DOCTYPE html>");
  myWindow.document.writeln("<html><head><title>New Window</title></head><body>");

  myWindow.document.writeln("<h1 id='heading1'>Hi from
JavaScript</h1></body></html>");
  myWindow.document.close();
  myWindow.focus();
}

window.onload = function() {
  document.getElementById("addButton").onclick = domWindowAdd;
  document.getElementById("openButton").onclick = createWindow;
};
</script>
</head>
<body>
<h1>DOM Window Interaction</h1>
<form>
  <input type="text" id="textToAdd" size="30" value="New Text">
  <input type="button" value="Add Text" id="addButton">
  <input type="button" value="Open Window" id="openButton">
</form>
</body>
</html>
```

ONLINE <http://javascriptref.com/3ed/ch12/windowadd.html>

Full-Screen Windows

Creating a window that fills up the screen and even removes browser chrome is possible in many browsers. It has long been possible to figure out the current screen size by consulting `window.screen` properties and then create a new window that fits most or all of the available area. The script fragment presented here should work to fill up the screen in all modern browsers:

```
var newwindow=window.open("http://www.google.com","main",  
"height="+screen.height+",width="+screen.width+",screenX=0,  
screenY=0,left=0,top=0,resizable=no");
```

The previous “poor man’s” script does keep the browser chrome and may not quite fill up the window. Under Internet Explorer and potentially other browsers, it may be possible to go into a full-screen mode more directly, using a feature string value, like so:

```
newWindow=window.open("http://www.google.com", "main","fullscreen=yes");
```

ONLINE <http://javascriptref.com/3ed/ch12/fullscreen.html>

Some archaic browsers needed a more complicated script and even prompted the user if a security privilege should be granted to go full screen. The fact that older browsers warned users before going full screen is quite telling, especially once you consider that some users will not know how to get out of full-screen mode. The key combination ALT+F4 should do the trick on a Windows system. However, users may not know this, so you should provide a Close button or instructions for how to get out of full-screen mode.

Firefox offers a `window.fullScreen` property that holds a Boolean indicating whether or not the window is in full-screen mode.

Centering Windows

Even things that should be easy with JavaScript windows are not that easy. For example, if we try to center a spawned Window, we might

be tempted to center it to the screen. That is fairly straightforward if we consult the screen dimensions found in the Screen object:

```
// given width and height contain the window sizes
var left = (screen.width/2)-(width/2);
var top  = (screen.height/2)-(height/2);

theNewWindow = window.open("http://www.google.com", "", "width=" +
width + ",height=" + height + ",top=" + Math.round(top) +
",left=" + Math.round(left));
```

However, you are more likely to want to center a spawned window in relation to the spawning Window. This becomes a bit trickier because of the way the inner and outer browser Window dimensions are calculated and because of the variation in a browser's toolbar and button heights. The following example shows roughly what you would do:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>center window.open()</title>
</head>
<body>
<h1>center window.open()</h1>
<form>
<label>Height:<input type="text" id="height" size="4" maxlength="4"
value="300"></label><br>
<label>Width:<input type="text" id="width" size="4" maxlength="4"
value="300"></label><br><br>
<input type="button" value="Create Window" id="openButton">
<input type="button" value="Close Window" id="closeButton">
</form>
<script>
function openWindow() {
    var browserWH = getBrowserWH();
    var browserXY = getBrowserXY();
    var width = document.getElementById("width").value;
    var height = document.getElementById("height").value;
    var left = (browserWH[0]/2)-(width/2) + browserXY[0];
    var top = (browserWH[1]/2)-(height/2) + browserXY[1];
    theNewWindow = window.open("http://www.google.com", "", "width="
    + width + ",height=" + height + ",top=" + top + ",left=" + left);
}
```

```

function getBrowserWH() {
    var width, height;
    if (window.innerWidth){
        width = window.innerWidth;
        height = window.innerHeight;
    }
    else if (document.documentElement && document.documentElement.clientWidth) {
        width = document.documentElement.clientWidth;
        height = document.documentElement.clientHeight;
    }
    return [width,height];
}

function getBrowserXY() {
    var x,y;
    if ("screenX" in window) {
        x = window.screenX;
        y = window.screenY;
    }
    else if ("screenLeft" in window) {

        x = window.screenLeft;
        y = window.screenTop
    }

    return [x,y];
}

function closeWindow() {
    if (window.theNewWindow)
        theNewWindow.close();
}

window.onload = function() {
    document.getElementById("openButton").onclick = openWindow;
    document.getElementById("closeButton").onclick = closeWindow;
};
</script>
</body>
</html>

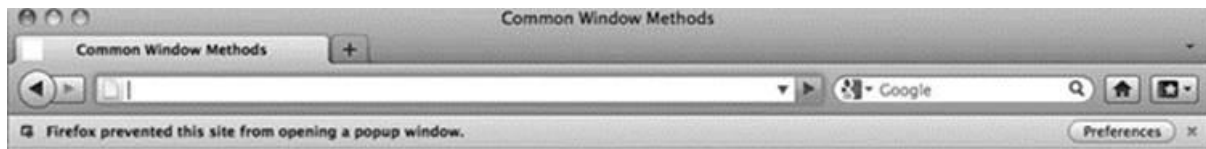
```

ONLINE <http://javascriptref.com/3ed/ch12/windowcenter.html>

It seems that the more you try to make spawned windows work, the more trouble you discover. We save the final straw before presenting the “solution” most developers adopt.

Pop-up Blockers

The most significant drawback to spawned windows is that many browsers may kill their load because of the abuse of so-called pop-ups online for advertising and other purposes. Generally, if a window is created on page load, timer, or without some user-initiated action, the browser will try to block it, as shown here:



Obviously, if you are counting on a window being raised for some custom dialog or important functionality, a pop-up blocker can be quite troublesome. This leads developers to question whether a pop-up blocked can be detected. The answer, unfortunately, is "sometimes." The general method to pop-up blocking detection is to try to open a pop-up window on load and then see if it exists or if it has the closed property set to true. Note that even pop-up blockers allow pop-ups when there is user interaction, so the test must be done without the input from the user.

Unfortunately, this isn't a perfect solution. Browsers vary in how they address pop-ups. For example, Chrome opens the window and hides it, so this test fails for Chrome. The accepted method in Chrome is to wait a little bit and then check the innerWidth. If it is blocked, it will be 0. You do have to wait a bit because right on pop-up load the innerWidth is still set to 0, even when pop-ups are allowed. In addition, if the user manually allows the pop-up, it likely will not be detected properly, as the detection code will run before the pop-up is launched. Other issues will likely emerge as time passes. A simple example showing this less-than-optimal detection scheme is given here:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Popup Blocker Test</title>
<script>
var popup;
function createWindow() {
    popup = window.open("popup.html", "", "height=1,width=1");
    if (!popup || popup.closed || typeof popup.closed=="undefined") {
        document.getElementById("message").innerHTML = "Popup Blocked";
        return;
    }
    popup.blur();
    window.focus();
    setTimeout(checkChrome, 100);
}

function checkChrome() {
    if (popup.innerHeight > 0) {
        document.getElementById("message").innerHTML = "Popup Allowed";
    }
    else {
        document.getElementById("message").innerHTML = "Popup Blocked";
    }
    popup.close();
}

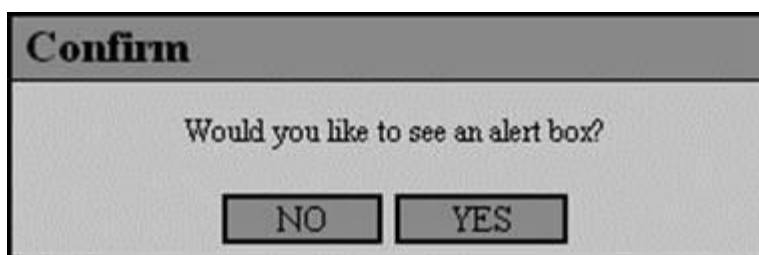
window.onload = function() {
    createWindow();
};
</script>
</head>
<body>
<div id="message"></div>
</body>
</html>
```

ONLINE <http://javascriptref.com/3ed/ch12/popupdetector.html>

Besides all the browser quirks, the pop-up detection scheme is quite imperfect because we can see the detection window. There are just too many problems with spawned windows because of past abuse, so many developers have left them in favor of CSS-based overlays.

Overlays Instead of Windows

Unfortunately, as we have seen, simple dialogs such as `alert()` and `prompt()` lack customization. You may opt to try to create custom dialogs using the generic `window.open()` method. However, in either case, the dialogs may be blocked by browser-based or third-party pop-up blockers installed by the user. To address both the customization concerns and pop-up blockers, many designers have turned to what we dub “div dialogs,” named for the HTML `<div>` tag used to create them. Using CSS, designers can position `<div>` tag–based regions over content and customize them visually in whatever manner they like.



The creation of a div dialog follows standard DOM tag-building code. First, the `<div>` tag that would be used as the custom dialog would be created and positioned:

```
var dialog = document.createElement("div");
dialog.className = "center";
```

Then the various elements would be added to the dialog, typically one at a time, unless you resort to using the `innerHTML` property:

```
var dialogTitle = document.createElement("h3");
var dialogTitleText = document.createTextNode("Warning!");
dialogTitle.appendChild(dialogTitleText);
dialog.appendChild(dialogTitle);
// etc.
```

We show only a snippet here because it gets quite lengthy as the messages and various controls to dismiss the dialog are added, and the repetitious code adds little to the discussion. Once performed, though, the procedure can be abstracted into a custom function such as `createDialog()`, where you could indicate the type of dialog, message, and style needed.

After all of the various elements have been added to the dialog, the region is displayed at the desired page position. However, there is one important consideration we need to mention before pointing readers to the complete example online: the issue of modality. Normally, `alert()` and `confirm()` dialogs are application modal, meaning that the user must address them before moving on to another browser-based activity. To simulate modality, we create a translucent region that covers the browser window or region we want to be modal to. To do this first, create a `<div>` tag to serve as the modality overlay:

```
function createOverlay() {
    var div = document.createElement("div");
    div.className = "grayout";
    document.body.appendChild(div);
    return div;
}
```

Now, make sure the appropriate CSS is applied to make the overlay translucent and covering the region to shield from user activity. The class name set in the preceding function does this and is shown here as reference:

```
.grayout {
    position: absolute;
    z-index: 50;
    top: 0px; left: 0px;
    width: 100%; height: 100%;
    filter:alpha(opacity=80);
    -moz-opacity: 0.8;
    opacity: 0.8;
    background-color: #999;
    text-align: center;
}
```

Finally, append it in the document along with the dialog, as shown here:

```
var parent = document.createElement("div");
parent.style.display = "none";
parent.id = "parent";
var overlay = createOverlay();
overlay.id = "overlay";
parent.appendChild(overlay);

var dialog = createDialog(type, message);
/* assume type and message are used to build
   a particular type of dialog with the passed
   message */
parent.appendChild(dialog);

document.body.appendChild(parent);
parent.style.display = "block";
```

A simple example demonstrating simple `<div>`-based dialogs is shown here and previewed in *Figure 12-2*:



Figure 12-2 Overlay in action

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Simple Overlay Dialog</title>
<style>
html,body{height:100%;}

.center{position:absolute;
left:50%;
top: 30px;
width:300px;
margin-top:50px;
margin-left:-116px;
border:2px solid #000;
background-color:#ccf;
z-index: 200;
text-align:center;
}

.grayout{position: absolute;
z-index: 50;
top: 0px;
left: 0px;
width: 100%;
height: 100%;
filter:alpha(opacity=80);
-moz-opacity: 0.8;
opacity: 0.8;
font-size: 70pt;
background-color: #999;
text-align: center;
}
</style>
<script>

function createDialog(modal) {
var parent = document.createElement("div");
parent.style.display = "none";
parent.id = "parent";

if (modal) {
var overlay = createOverlay();
overlay.id = "overlay";
parent.appendChild(overlay);
}

var dialog = document.createElement("div");
dialog.className = "center";

var windowHTML = "<h1 style='align:center; border-bottom:

```



```

lpx solid black; margin-bottom: 0'>";
    windowHTML += "Overlay Dialog</h1><div style='align:center;
background: white'><form>";
    windowHTML += "<br><input type='button' value='CLOSE'
onclick='removeDialog();'>";
    windowHTML += "</form></div>";

    dialog.innerHTML = windowHTML;
    parent.appendChild(dialog);

    document.body.appendChild(parent);
    parent.style.display = "block";
}

function removeDialog() {
    var parent = document.getElementById("parent");
    document.body.removeChild(parent);
    return false;
}

function createOverlay() {
    var div = document.createElement("div");
    div.className = "grayout";
    document.body.appendChild(div);
    return div;
}

window.onload = function() {
    document.getElementById("dialogButton").onclick = function() {
createDialog(true); };
    document.getElementById("modelessDialogButton").onclick =
function() { createDialog(false); };
};
</script>
</head>
<body>
<h3>Overlay Dialogs</h3>
<form>
    <input type="button" id="dialogButton" value="Open Modal Dialog">
    <input type="button" id="modelessDialogButton" value="Open Modeless Dialog">
</form>
</body>
</html>

```

ONLINE <http://javascriptref.com/3ed/ch12/overlay.html>

Note that our aim in this section is not to provide a complete solution with different dialog types, return values, styles, and the like. This would be best suited for a library. Our intention here is to show the technique only.

Controlling Windows

As we have seen so far, it is easy enough to open and close windows as well as write content to them. There are numerous other ways to control windows.

focus() and blur()

It is possible to bring a window to focus using the `window.focus()` method. This should raise the window for access. Conversely, it is also possible to do the opposite using the `window.blur()` method.

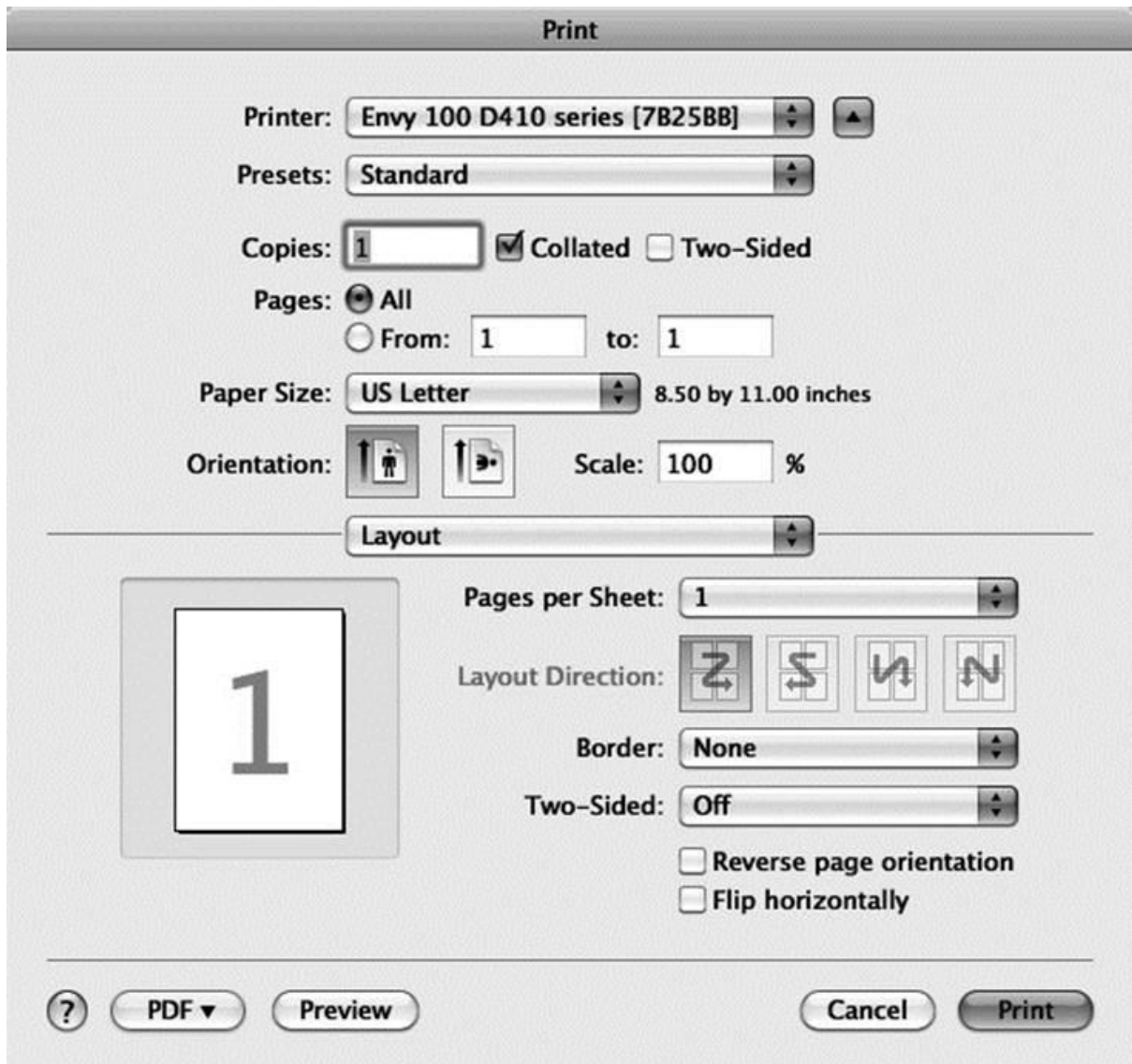
NOTE There may be security considerations in Internet Explorer that may cause a window not to respect a `focus()` invocation. Test your browser or consult the Microsoft Developer Network (MSDN) for the latest information, as it changes between versions of Internet Explorer.

stop()

Some methods of window control address common browser functions. For example, if a window is taking a long time to load, an end user may hit the Stop button. This can be accomplished programmatically with the `window.stop()` method.

print()

The HTML5 specification standardizes `window.print()`, which has long been supported by browsers. Firing this method should raise a dialog first, like so:



find()

Some browsers implement the nonstandard `window.find()` method. The syntax of this method is historically written to be

`window.find(targetstring, casesensitivity, backwards, wraparound, wholeword, searchinframes, showdialog)`

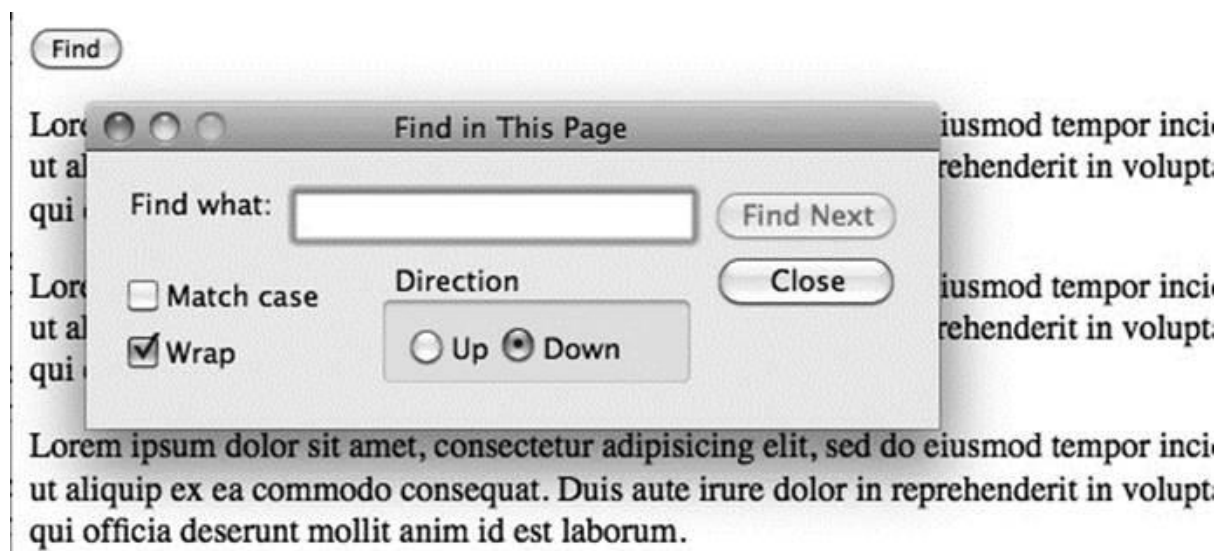
where

- *targetstring* is the string to find.
- *casesensitivity* is a Boolean value that, if true, indicates that the search should be performed with case sensitivity.

- *backwards* is a Boolean value that, if true, indicates that the search should proceed backward rather than forward.
- *wraparound* is a Boolean value that, if true, indicates that the search should wrap to the top of the document once the bottom is hit.
- *wholeword* is a Boolean value that, if true, indicates that the search should only match whole words.
- *searchinframes* is a Boolean value that, if true, indicates that the contents of frames within the window should be searched.
- *showdialog* is a Boolean value that, if true, shows the browser's search dialog.

The reality is that, generally, this isn't the case. However, some browsers will support a simple invocation of `window.find()` to pop the browser's find command, as shown here:

```
<form>
  <input type="button" value="Find" id="findBtn" onclick="window.find();" />
</form>
```



Given the eventual coverage of printing, this actually doesn't seem a long shot to be eventually codified and more widely supported, though admittedly that is still speculation at this point.

NOTE There are a few other possibilities for browser-related window actions such as adding bookmarks or even trying to programmatically set the home page. However, these are not only nonstandard but also poorly supported.

Moving Windows

Moving windows around the screen is possible using two different methods, `window.moveBy()` and `window.moveTo()`. The `moveBy()` method moves a window a specified number of pixels and has a syntax of

```
windowname.moveBy(horizontalpixels, verticalpixels)
```

where

- *windowname* is the name of the window to move or is called just Window if it is the main window.
- *horizontalpixels* is the number of horizontal pixels to move the Window, where positive numbers move the window to the right and negative numbers to the left.
- *verticalpixels* is the number of vertical pixels to move the Window, where positive numbers move the window down and negative numbers up.

For example, given that a window called *myWindow* exists, the following would move the window down 100 pixels and to the right 100 pixels:

```
myWindow.moveBy(100, 100);
```

If you have a particular position in the screen to move a window to, it is probably better to use the `window.moveTo()` method, which will move a window to a particular x,y coordinate on the screen. The syntax of this method is

```
windowname.moveTo(x-coord, y-coord)
```

where

- *windowname* is the name of the window to move or is called `Window` if it is the main window.
- *x-coord* is the screen coordinate on the x-axis to move the window to.
- *y-coord* is the screen coordinate on the y-axis to move the window to.

So given that the window called *myWindow* is on the screen, the following would move the window to the origin of the screen:

```
myWindow.moveTo(1, 1);
```

Resizing Windows

In JavaScript, the methods for resizing windows are very similar to the ones for moving them. The method `window.resizeBy(horizontal, vertical)` resizes a window by the values given in *horizontal* and *vertical*. Negative values make the window smaller, while positive values make it bigger, as shown in the examples here:

```
myWindow.resizeBy(10, 10); // makes the window 10 pixels taller and wider  
myWindow.resizeBy(-100, 0); // makes the window 100 pixels narrower
```

Similar to the `moveTo()` method, `window.resizeTo(width, height)` resizes the window to the specified *width* and *height* indicated:

```
myWindow.resizeTo(100, 100); // make window 100x100  
myWindow.resizeTo(500, 100); // make window 500x100
```

NOTE In modern JavaScript implementations, it is not possible to resize browser windows to a very small size, say 1×1 pixels. This could be construed as a security hazard, as a user may not notice such a minuscule window spawned by a site after leaving it.

Scrolling Windows

Similar to resizing and moving, the Window object supports the scrollBy() and scrollTo() methods to correspondingly scroll a window by a certain number of pixels or to a particular pixel location. The following simple examples illustrate how these methods might be used on some window called *myWindow*:

```
myWindow.scrollBy(10, 0); // scroll 10 pixels to the right
myWindow.scrollBy(-10, 0); // scroll 10 pixels to the left
myWindow.scrollBy(100, 100); // scroll 100 pixels to the right and down
myWindow.scrollTo(1, 1); // scroll to the origin of 1, 1
myWindow.scrollTo(100, 100); // scroll to 100, 100
```

NOTE The method scroll() may occasionally be encountered. While the syntax of scroll() is identical to scrollBy(), the method is nonstandard and should be avoided. In addition, scrollByLines(*lines*) and scrollByPages(*pages*) are two similar methods supported only by Firefox. The same effect can be achieved with scrollBy(), so again we recommend using only scrollBy().

In addition to scrolling the Window, it is often desirable to see where the browser has been scrolled to. Different actions may occur, depending on where the user is on the page. However, finding the scroll location is quite different depending on the browser. The most obvious properties to look at would be scrollX and scrollY. However, these properties are not supported by Opera and Internet Explorer. The pageXOffset and pageYOffset properties are now supported in all major browsers, including Internet Explorer from version 9. In order to get the scroll position before version 9, it is necessary to look at the document.documentElement.scrollLeft and document.documentElement.scrollTop properties:

```

function getScrollPosition() {
    var scrollX, scrollY;
    if ("pageXOffset" in window) {
        scrollX = window.pageXOffset;
        scrollY = window.pageYOffset;
    }
    else {
        scrollX = window.document.documentElement.scrollLeft;
        scrollY = window.document.documentElement.scrollTop;
    }
}
}

```

A complete example presented here can be used to experiment with the various common Window methods that we have discussed here:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Common Window Methods</title>
<script>
var myWindow; // note global here, wrap in object if you like

window.addEventListener("load", function () {

function openWindow() {
    myWindow = open("", "mywin", "height=400, width=500, scrollbars=yes");

    var HTMLstr = "<!DOCTYPE html>";
    HTMLstr += "<html>\n<head>\n<meta charset='utf-8'>\n<title>Test Window</title>\n\n</head>\n";
    HTMLstr += "<body style='background: #fc6'>\n<h1>JavaScript Window Methods\n</h1>\n";
    HTMLstr += "<div style='height=600px;width=600px;background:#fc6'>\n<img\nsrc='HTML5.png'>\n</div>";
    HTMLstr += "</body></html>";

    myWindow.document.writeln(HTMLstr);
    myWindow.document.close();
    myWindow.focus();
}

function moveWindowTo() {
    if ((window.myWindow) && (myWindow.closed == false))
        myWindow.moveTo(document.getElementById("moveX").value, document.
getElementById("moveY").value);
}

```

```

function moveWindowBy(x, y) {
  if ((window.myWindow) && (myWindow.closed == false))
    myWindow.moveBy(x, y);
}

function scrollWindowTo() {
  if ((window.myWindow) && (myWindow.closed == false))
    myWindow.scrollTo(document.getElementById("moveX").value,
    document.getElementById("scrolly").value);
}

function scrollWindowBy(x,y) {
  if ((window.myWindow) && (myWindow.closed == false))
    myWindow.scrollBy(x, y);
}

function getScrollPosition() {
  var scrollX, scrollY;
  if ((window.myWindow) && (myWindow.closed == false)){

    if ("pageXOffset" in window) {
      scrollX = myWindow.pageXOffset;
      scrollY = myWindow.pageYOffset;
    }

    else {
      scrollX = myWindow.document.documentElement.scrollLeft;
      scrollY = myWindow.document.documentElement.scrollTop;
    }
    var message = "scrollX: " + scrollX + "/ scrollY: " + scrollY;
    document.getElementById("scrollMessage").innerHTML = message;
  }
}

function resizeWindowTo() {
  if ((window.myWindow) && (myWindow.closed == false))
    myWindow.resizeTo(document.getElementById("resizeX").value,
    document.getElementById("resizeY").value);
}

function resizeWindowBy(x,y) {
  if ((window.myWindow) && (myWindow.closed == false))
    myWindow.resizeBy(x, y);
}

```



```

function closeWindow() {
    if (myWindow)myWindow.close();
}

function focusWindow() {
    if (myWindow)myWindow.focus();
}

function blurWindow() {
    if (myWindow) myWindow.blur();
}

function stopWindow() {
    if (myWindow) myWindow.stop();
}

function printWindow() {
    if (myWindow) myWindow.print();
}

document.getElementById("openBtn").addEventListener("click", openWindow, true);
document.getElementById("closeBtn").addEventListener("click", closeWindow, true);
document.getElementById("focusBtn").addEventListener("click", focusWindow,
true);
document.getElementById("blurBtn").addEventListener("click", blurWindow, true);
document.getElementById("stopBtn").addEventListener("click", stopWindow, true);
document.getElementById("printBtn").addEventListener("click", printWindow, true);

document.getElementById("upMoveBtn").addEventListener("click",
function() { moveWindowBy(0, -10); }, true);
document.getElementById("leftMoveBtn").addEventListener("click",
function() { moveWindowBy(-10, 0); }, true);
document.getElementById("rightMoveBtn").addEventListener("click",
function() { moveWindowBy(10, 0); }, true);
document.getElementById("downMoveBtn").addEventListener("click", function() {
moveWindowBy(0, 10); }, true);

document.getElementById("moveBtn").addEventListener("click",
moveWindowTo, true);

```

```

document.getElementById("upScrollBtn").addEventListener("click", function() {
scrollWindowBy(0, -10); }, true);

document.getElementById("leftScrollBtn").addEventListener("click",
function() { scrollWindowBy(-10, 0); }, true);

document.getElementById("rightScrollBtn").addEventListener("click",
function() { scrollWindowBy(10, 0); }, true);

document.getElementById("downScrollBtn").addEventListener("click",
function(){ scrollWindowBy(0, 10); }, true);

document.getElementById("scrollBtn").addEventListener("click",
scrollWindowTo, true);
    document.getElementById("upResizeBtn").addEventListener("click",
function(){ resizeWindowBy(0, -10); }, true);
    document.getElementById("leftResizeBtn").addEventListener("click",
function(){ resizeWindowBy(-10, 0);}, true);
    document.getElementById("rightResizeBtn").addEventListener("click",
function(){ resizeWindowBy(10, 0); }, true);
    document.getElementById("downResizeBtn").addEventListener("click",

function() { resizeWindowBy(0, 10); }, true);
    document.getElementById("resizeBtn").addEventListener("click",
resizeWindowTo, true);
    openWindow();
}, true);

</script>
</head>
<body>
<h1>Window Methods Tester</h1>
<hr>
<form name="testform" id="testform">

<input type="button" value="Open Window" id="openBtn">
<input type="button" value="Stop Load" id="stopBtn">
<input type="button" value="Close Window" id="closeBtn">
<input type="button" value="Focus Window" id="focusBtn">
<input type="button" value="Blur Window" id="blurBtn">
<input type="button" value="Print" id="printBtn">
<br><br>
<input type="button" value="Move Up" id="upMoveBtn">
<input type="button" value="Move Left" id="leftMoveBtn">
<input type="button" value="Move Right" id="rightMoveBtn">
<input type="button" value="Move Down" id="downMoveBtn">
<br><br>

```

```
<label>X: <input type="text" size="4" id="moveX" value="0"></label>
<label>Y: <input type="text" size="4" id="moveY" value="0"></label>
<input type="button" value="Move To" id="moveBtn">
<br><br>
<input type="button" value="Scroll Up" id="upScrollBtn">
<input type="button" value="Scroll Left" id="leftScrollBtn">
<input type="button" value="Scroll Right" id="rightScrollBtn">
<input type="button" value="Scroll Down" id="downScrollBtn">
<br><br>

<label>X: <input type="text" size="4" id="scrollX" value="0"></label>
<label>Y: <input type="text" size="4" id="scrollY" value="0"></label>
<input type="button" value="Scroll To" id="scrollBtn">
<br>
<div id="scrollMessage"></div>
<br><br>

<input type="button" value="Resize Up" id="upResizeBtn">
<input type="button" value="Resize Left" id="leftResizeBtn">
<input type="button" value="Resize Right" id="rightResizeBtn">
<input type="button" value="Resize Down" id="downResizeBtn">
<br><br>
<label>X: <input type="text" size="4" id="resizeX" value="0"></label>
<label>Y: <input type="text" size="4" id="resizeY" value="0"></label>
<input type="button" value="Resize To" id="resizeBtn">
<br><br>

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

ONLINE <http://javascriptref.com/3ed/ch12/windowmethods.html>

An example rendering of the previous example is shown in *Figure 12-3*.

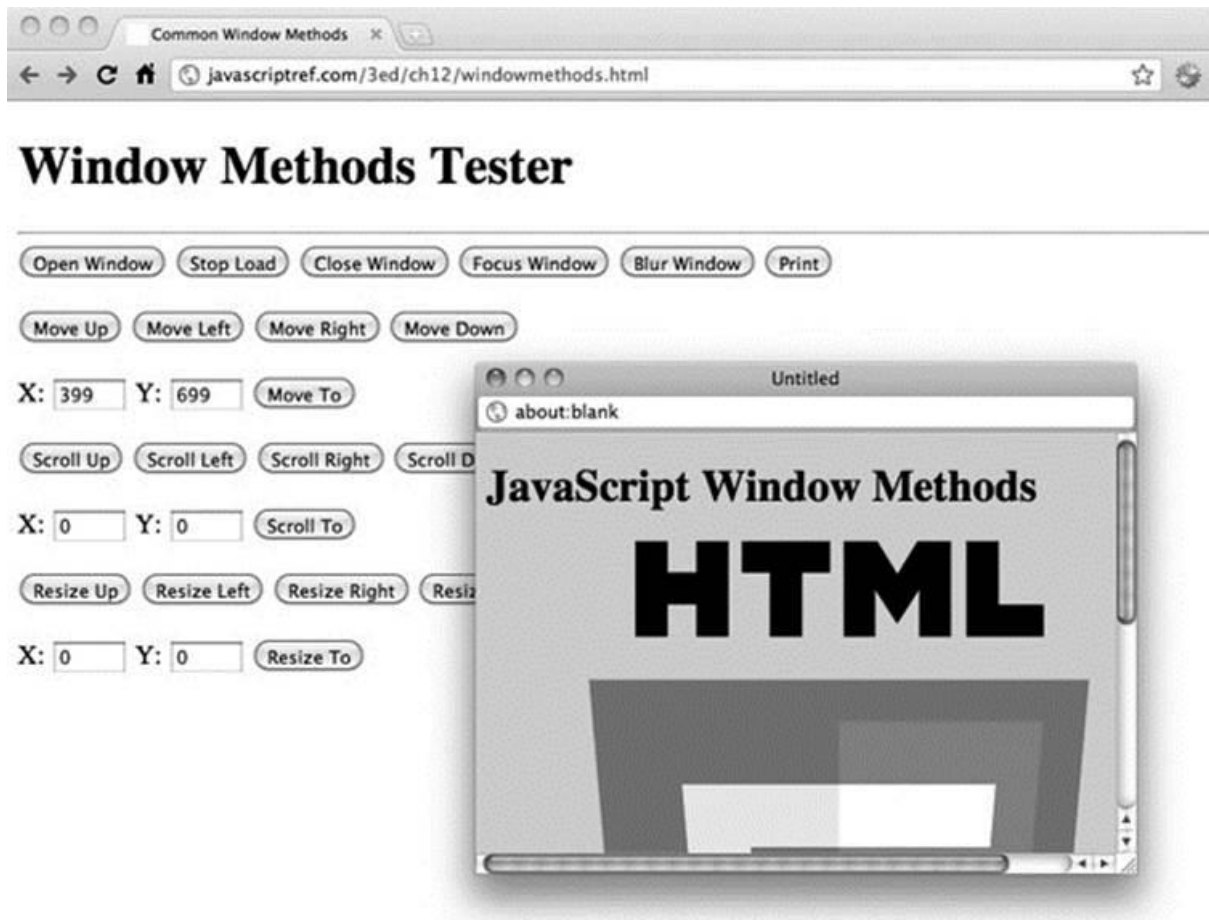


Figure 12-3 Controlling standard spawned windows

Accessing and Setting a Window's Location

It is often desirable to set a window to a particular URL. There are numerous ways to do this in JavaScript, but the best way is to use the Location object, which is a property of Window. The Location object is used to access the current location (the URL) of the window. The Location object can be both read and replaced, so it is possible to update the location of a page through scripting. The following example shows how a simple button click can cause a page to load:

```
<form>
  <input type="button" value="Go to Google"
    onclick="window.location='http://www.google.com';">
</form>
```

Rather than direct assignment, you can use the assign() method as well.

```
<form>
  <input type="button" value="Go to Google"
    onclick="window.location.assign('http://www.google.com');">
</form>
```

NOTE Internet Explorer defined the nonstandard `window.navigate(URL)`, which will load the specified URL parameter, similar to setting the location value. This is a nonstandard method that should be avoided, though it is also supported in some other browsers, notably Opera.

Regardless of the method used to change the URL, the new location will be added to the browser history. If you desire to replace the current page in history, use the `replace()` method:

```
<form>
  <input type="button" value="Go to Google (no back button)"
    onclick="window.location.replace('http://www.google.com');">
</form>
```

If you just want to refresh the page without setting the URL, you may use the `reload()` method:

```
<form>
  <input type="button" value="Reload Page" onclick="window.location.reload();">
</form>
```

A complete list of the methods is shown in *Table 12-4*.

Table 12-4 Location Methods

Method	Description
<code>assign(URL)</code>	Change the location of the current page with the passed in <i>URL</i> .
<code>reload()</code>	Reload the current page.
<code>replace(URL)</code>	Replaces the current page with the given <i>URL</i> in history. As it is replaced in history, it won't be possible to access the current page with back/forward.

It is also possible to access parsed pieces of the Location object to see where a user is at a particular moment. A few examples are shown here:

```
alert(window.location.protocol); // shows the current protocol in the URL
alert(window.location.hostname); // shows the current hostname
alert(window.location.href); // shows the whole URL
```

The properties of the Location object are pretty straightforward for anyone who understands a URL. A complete list of these properties can be found in *Table 12-5*.

Table 12-5 Common Properties of the Location Object

Property	Description
hash	The part of the URL including and following the # symbol.
host	The hostname and port number.
hostname	The hostname.
href	The entire URL.
pathname	The path relative to the host.
port	The port number.
protocol	The protocol of the URL.
search	The part of the URL including and after the ?.

One property that bears some attention is search. This property contains the query string, and very likely you will want to break it up into its constituent name-value pairs. Some simple string manipulations will do the trick:

```
var pairStr = "";
var queryString = window.location.search.substring(1);
var pairs = queryString.split("&");
for (var i=0; i < pairs.length; i++) {
    var kv = pairs[i].split("=");
    pairStr += "Key: " + kv[0] + " Value: " + kv[1] + "\n";
}
alert(pairStr);
```

Finally, an emerging method called `resolveURL()` is specified under HTML5. This method returns the absolute path of a relative URL passed in. For example,

```
// given current URL is http://javascriptref.com/3ed/ch12
alert(location.resolveURL("../../index.html"));
```

would show a dialog with the URL `http://javascriptref.com/index.html`.

NOTE No browser has implemented this method at the time of this edition's writing.

To conclude the section, we present an example demonstrating the various properties and methods of the Location object:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>window.location</title>
</head>
<body>
<h2>window.location</h2>
<form>
  <input type="button" id="infoBtn" value="Get Location Info">
  <input type="button" id="queryBtn" value="Get Query String Pairs">
  <input type="button" id="reloadBtn" value="Reload">
  <input type="button" id="assignBtn" value="Go to Google">
  <input type="button" id="replaceBtn" value="Replace with Google">
  <input type="button" id="resolveBtn" value="Get Absolute URL">
</form>
<br><br>
<div id="message"></div>

<script>
window.addEventListener("load", function () {
  document.getElementById("infoBtn").addEventListener("click", function () {
    var str = "Href: " + window.location.href + "<br>";
    str += "Protocol: " + window.location.protocol + "<br>";
    str += "Host: " + window.location.host + "<br>";
    str += "Hostname: " + window.location.hostname + "<br>";
    str += "Port: " + window.location.port + "<br>";
    str += "Pathname: " + window.location.pathname + "<br>";
    str += "Search: " + window.location.search + "<br>";
    str += "Hash: " + window.location.hash + "<br><br>";
    document.getElementById("message").innerHTML += str;
  }, true);

  document.getElementById("reloadBtn").addEventListener("click",
function () { window.location.reload(); }, true);
  document.getElementById("assignBtn").addEventListener("click",
function () { window.location.assign("http://www.google.com"); }, true);
  document.getElementById("replaceBtn").addEventListener("click",
function () { window.location.replace("http://www.google.com"); }, true);
  document.getElementById("resolveBtn").addEventListener("click",
function () {
```

```

var path = "checkurl.html";
var absolutePath = window.location.resolveURL(path);
document.getElementById("message").innerHTML +=
"Path: " + path + " Absolute Path: " + absolutePath + "<br><br>";
}, true);
document.getElementById("queryBtn").addEventListener("click", function ()
{
    var queryString = window.location.search.substring(1);
    var pairs = queryString.split("&");
    for (var i=0; i<pairs.length; i++) {
        var kv = pairs[i].split("=");
        document.getElementById("message").innerHTML +=
        "Key: " + kv[0] + " Value: " + kv[1] + "<br>";
    }
}, true);

// show initial load message
document.getElementById("message").innerHTML =
"This page was loaded at : " + (new Date()).toString() + "<br><br>";
}, true);
</script>
</body>
</html>

```

ONLINE <http://javascriptref.com/3ed/ch12/location.html>

Hash Values in URLs

One aspect of URLs that deserves a mention is the hash portion that specifies the fragment identifier of the page. Reading the value is obviously quite easy.

```
alert(window.location.hash);
```

Setting the hash is just as easy.



Set hash



```
<a href="javascript: window.location.hash = 'smoked';">Set hash</a>
```



Set hash

Obviously, this will change the hash, but notice that the Web browser does not refresh while the hash value typically creates an entry in the browser's history.

NOTE In some older browsers, hash changes were not handled properly when screen refreshes did not happen; the history was not actually affected, which defeats the purpose of “fixing” the back button action.

The nonrefreshing behavior of the fragment identifier URL is quite useful, as it allowed Web developers building Ajax and Flash applications to push a change to the URL for history and bookmark management without a page reload. HTML5 codifies this action by

adding anonhashchange event to easily signal the potential change of state. An example of this is demonstrated here:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>onhashchange</title>
</head>
<body>
<h1>onhashchange</h1>
<p>Enter names and then use the back/forward navigation to see that the message
is updated based on the hash.</p>
<form>
  <input type="text" id="name">
  <input type="button" value="Set Name" id="nameBtn"><br>
</form>

<div id="message"></div>
<script>
function setName() {
  var name = document.getElementById("name").value;
  window.location.hash = name;
  document.getElementById("name").value = "";
}

function updateName() {
  var name = document.location.hash.substring(1);
  if (name != "") {
    document.getElementById("message").innerHTML = "Hello " + name;
  }
  else{
    document.getElementById("message").innerHTML = "";
  }
}

window.onload = function() {
  updateName();
  document.getElementById("nameBtn").onclick = setName;
  window.onhashchange = updateName;
};
</script>
</body>
</html>
```

A more appropriate way to handle statement management is provided by HTML5 with the `pushstate()` and `replacestate()` methods of the History object, which are discussed next.

Manipulating a Window's History

When users press their browser's Back or Forward button, they are navigating the browser's history list. JavaScript provides the History object as a way to access the history list for a particular browser window. The History object is a read-only array of URL strings that show where the user has been recently. The main methods allow forward and backward progress through the history, as shown here:

```
<a href="javascript: window.history.forward();">Forward</a>  
<a href="javascript: window.history.back();">Back</a>
```

NOTE You should be careful when trying to simulate the Back button with JavaScript, as it may confuse users who expect links in a page labeled "Back" not to act like the browser's Back button.

It is also possible to access a particular item in the history list relative to the current position using the `history.go()` method. Using a negative value moves to a history item previous to the current location, while a positive number moves forward in the history list. For example:

```
<a href="javascript: window.history.go(-2);">Back two times</a>  
<a href="javascript: window.history.go(3);">Forward 3 times</a>
```

Given that it is possible to read the length of the `history[]` array using the `history.length` property, you could easily move to the end of the list using the following:

```
<a href="javascript: window.history.go(window.history.length-1);">Last Item</a>
```

Direct access to the URL elements in the history is not possible with JavaScript; in the past, however, unscrupulous individuals have shown that calculated guesses of URLs in conjunction with the rendered styles of visited links can reveal past browsing habits. A simple example of a less nefarious use of the History object can be found online.

ONLINE <http://www.javascriptref.com/3ed/ch12/history.html>

pushstate() and replacestate()

The rise of Web applications and Ajax require much more programmer intervention in history management than in the past. On the Web, traditionally each unique URL represented a unique page or state of the Web application. However, in an Ajax-style application, often this is not the case. In order not to break the Back button and other Web semantics such as bookmarking, ingenious developers discovered that they could use the hash value to indicate a state change because it did not cause a browse screen refresh. The HTML5 specification attempts to ease this transition with the introduction of the window.pushState() and window.replaceState() methods.

The syntax of the pushState() method is

```
pushState(stateObject, title [, URL])
```

where

- *stateObject* is a JSON structure containing the information to save.
- *title* is the title for the browser's title bar and/or history list.
- *URL* is the URL to display in the browser's location, though there is not a network load related to this, so the URL can be arbitrary.

When pushState() is called, it changes the browser's URL to the passed-in *URL*. This will not necessarily be related to a network load; however, the newly set URL will be used in the Location object as will the Referer header on network requests. After being set, a future use of the browser's Back or Forward button will fire the window.onpopstate event and will receive the saved state object.

The syntax of the replaceState() method is pretty much the same:

```
replaceState(stateObject, title [, URL])
```

The only difference is that the *stateObject* replaces the current history item rather than making a new one. An example that can be used to explore these methods is shown here:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>pushState() and replaceState()</title>
<style>
  label {font-weight: bold; display: block;}
</style>
</head>
<body>
<h1>pushState() and replaceState()</h1>
<p>Enter information and then use the back/forward navigation to see
that the message is updated based on the hash.<br>
When you set the data the history item will be added.<br>
When you replace it the history will be modified.<br>
Notice that the URL does not contain the stored information.</p>

<form>
  <label for="name">Name:</label><input type="text" id="name">
  <label for="age">Age:</label><input type="text" id="age">
  <label for="food">Favorite Food:</label><input type="text" id="food">
  <br><br>
  <input type="button" value="Set User" id="setUserBtn">
  <input type="button" value="Replace User" id="replaceUserBtn">
</form>

<br><br>
<div id="message"></div>
<script>
var JSREF = {};
JSREF.setUser = function(replace){
  var name = document.getElementById("name").value;
  var age = document.getElementById("age").value;
  var food = document.getElementById("food").value;

  var user = {name: name, age: age, food: food, time: (new Date())};
  var id = (new Date()).getTime() + "a" + Math.floor(Math.random()*11);

  if (replace)
    window.history.replaceState(user, "User " + id, "user" + id + ".html");
  else
    window.history.pushState(user, "User " + id, "user" + id + ".html");

  // clear field values
  document.getElementById("name").value = "";
  document.getElementById("age").value = "";
  document.getElementById("food").value = "";
  JSREF.printUserInfo(user);
};
JSREF.updateUser = function(event){
  var user = event.state;
  if (user != null) {
    JSREF.printUserInfo(user);
  }
  else {
    document.getElementById("message").innerHTML = "";
  }
};

```

```

    }
};

JSREF.printUserInfo = function(user) {
    var str = "";
    str += "Name: " + user["name"] + "<br>";
    str += "Age: " + user["age"] + "<br>";
    str += "Favorite Food: " + user["food"] + "<br>";
    str += "Time Added: " + user["time"].toString() + "<br>";
    document.getElementById("message").innerHTML = str;
};

window.addEventListener("load", function(){

    document.getElementById("setUserBtn").addEventListener("click",
function () { JSREF.setUser(false); }, true);

    document.getElementById("replaceUserBtn").addEventListener("click",
function () { JSREF.setUser(true); }, true);

    window.onpopstate = JSREF.updateUser;
}, true);
</script>
</body>
</html>

```

ONLINE <http://javascriptref.com/3ed/ch12/pushreplacestate.html>

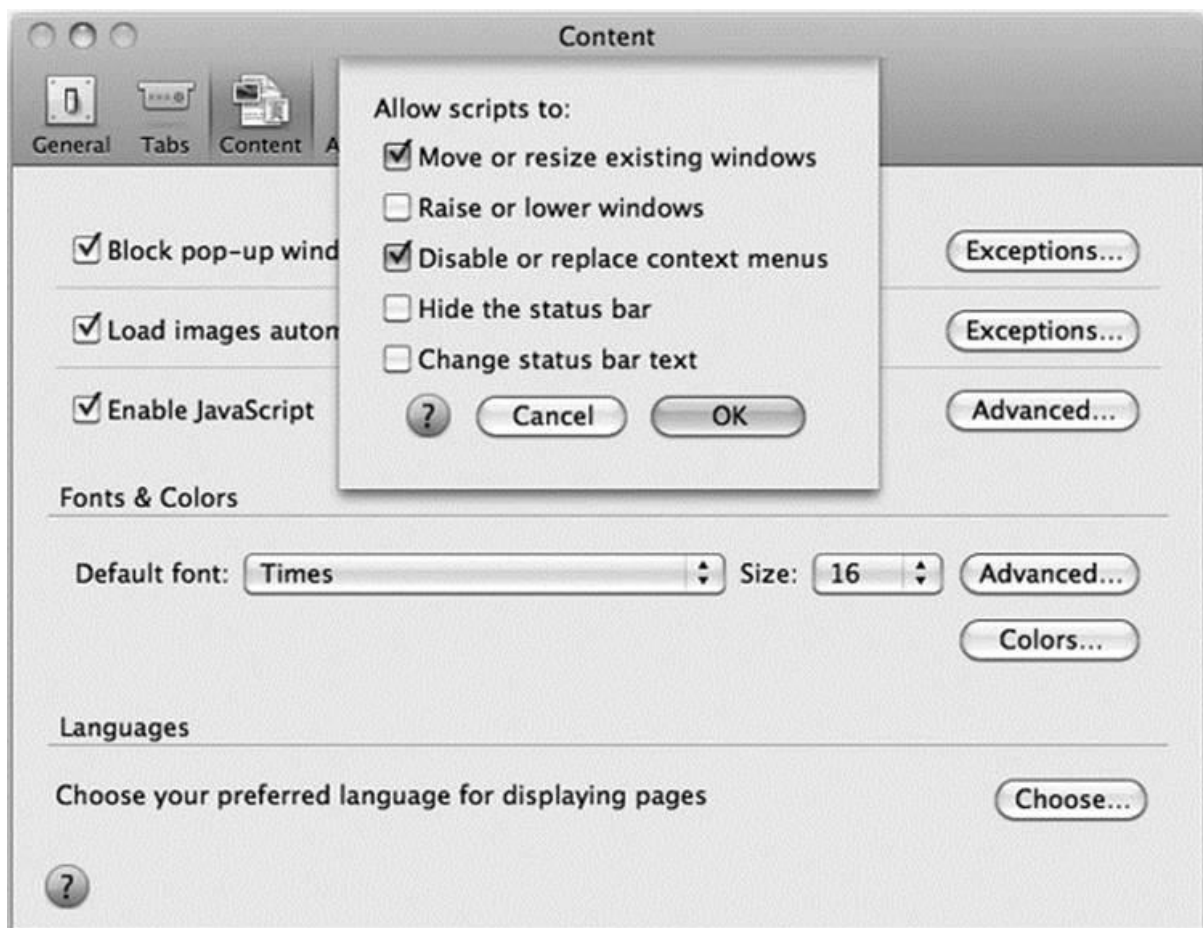
NOTE Browsers may save state values to the user's disk so they can be restored after the user restarts the browser. Because of this, there may be a size limit to the JSON representation of the user's state. For example, in the case of Firefox this limit is currently 640K characters. Saving state information beyond this would require the use of another mechanism such as `sessionStorage` or `localStorage`.

Trying to Control the Window's Status Bar

The status bar is the small text area in the lower-left corner of a browser window where messages are typically displayed, indicating download progress or other browser status items. Traditionally, it was possible to control the contents of this region with JavaScript. Many developers used this region to display short messages or even scrolling regions. The benefit of providing information in the status

bar is debatable, particularly when you consider the fact that manipulating this region often prevents default browser status information from being displayed—information which many users rely on.

Today the use of the status bar is quite limited, as many browsers simply do not show the status region anymore. In some browsers, it does not appear possible even to turn it on anymore. Even when the status bar can be seen, because of past abuse by phishers looking to trick end users, manipulation of the status bar via JavaScript is generally disallowed. As an example, note the advanced settings defaults for JavaScript in a recent version of Firefox showing this to be restricted:



The status bar can be accessed through two properties of the Window object: status and defaultStatus. The difference between these two properties is in how long the message is displayed. The value of defaultStatus is displayed any time nothing else is going on in a

browser window. The status value, on the other hand, is transient and is displayed only for a short period as an event (such as a mouse movement) happens. The simple example here exercises both properties:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>window.status and window.defaultStatus</title>
</head>
<body>
<h1>window.status and window.defaultStatus</h1>
<form>
<input type="button" value="set window.status" id="setStatusBtn">
<input type="button" value="show window.status" id="showStatusBtn">
<input type="button" value="set window.defaultStatus" id="setDefaultStatusBtn">
<input type="button" value="show window.defaultStatus" id="showDefaultStatusBtn">
</form>
<script>
window.onload = function () {

    document.getElementById("setStatusBtn").onclick = function () {
        window.status = "Standard status set";
    };

    document.getElementById("showStatusBtn").onclick = function () {
        alert(window.status);
    };

    document.getElementById("setDefaultStatusBtn").onclick = function () {
        window.defaultStatus = "Default status set";
    };

    document.getElementById("showDefaultStatusBtn").onclick = function () {
        alert(window.defaultStatus);
    };
    window.defaultStatus = "Default status set on page load";
};
</script>
</body>
</html>
```

ONLINE <http://www.javascriptref.com/3ed/ch12/status.html>

When trying the example, quite likely you may not see the status bar at all. Some browsers will return the values you set, but it will serve little practical purpose. Others will do nothing. Sadly, the status bar is but one example of a surgical removal of features in browsers to solve security or improve perceived usability. This implies that scripts may

work for some time before “rusting” away, as browser changes remove their value, so developers should aim to be aware of evolution of platforms, as it can affect their code.

Setting window Timeouts and Intervals

The Window object supports methods for setting timers that we might use to perform a variety of functions. These methods include `setTimeout()` and `clearTimeout()`. The basic idea is to set a timeout to trigger a piece of script to occur at a particular time in the future. The common syntax is

```
timerId = setTimeout(script-to-execute, time-in-milliseconds);
```

where

- *script-to-execute* is a string holding a function call or other JavaScript statement.
- *time-in-milliseconds* is the time to wait before executing the specified script fragment.

time-in-milliseconds has different minimum values depending on the browser, the method, and even the window’s active/inactive status. Notice that the `setTimeout()` method returns a handle to the timer that we may save in a variable, as specified by *timerId*. We might then clear the timeout (cancel execution of the function) later on using `clearTimeout(timerId)`. The following example shows how to set and clear a timed event:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>5,4,3,2,1...BOOM</title>
<script>
function startTimer() {
    timerId = setTimeout("window.close()", 5000);
    alert("Destruction in 5 seconds");
}

function stopTimer() {
    clearTimeout(timerId);
    alert("Aborted!");
}

window.onload = function() {
    document.getElementById("startBtn").onclick = startTimer;
    document.getElementById("stopBtn").onclick = stopTimer;
};
</script>
</head>
<body>
<h1>Browser Self-Destruct</h1>
<hr>
<form>
    <input type="button" value="Start Auto-destruct" id="startBtn">
    <input type="button" value="Stop Auto-destruct" id="stopBtn">
</form>
</div>
</body>
</html>

```

ONLINE <http://javascriptref.com/3ed/ch12/settimeout.html>

The `setInterval()` and `clearInterval()` methods should be used when a timed event occurs at a regular interval. Here is an example of the syntax of an interval:

```
var timer = setInterval("alert('When are we going to get there?')", 2000);
```

This example sets an alert that will fire every two seconds. To clear the interval, you would use a similar method as a timeout:

```
clearInterval(timer);
```

Now, quite often you will want to execute more than a bit of code in a timer or interval, and either method does allow you to pass in a function as you would with setting an event handler. This means that you can pass simply the function name, as shown here:

```
setTimeout(sayHello, 500);
```

In all browsers besides Internet Explorer, after the delay you can also pass parameters to pass to the later invoked function:

```
setTimeout(sayHello, 500, "Passed value!");
```

Likely you are not going to pass the parameters in this manner, as Internet Explorer browsers don't support it. Instead, we might pass parameters with a closure:

```
setTimeout(function(){sayHello(val);}, 500);
```

Of course, like any closure, we need to make sure we are careful that we get the value we want. For example, as with all closures, be careful inside of loops because the parameter will change as the loop goes on. The following example illustrates all of these points:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>setTimeout() with params</title>
<script>
function sayHello(counterStr) {
    if (!counterStr) {
        counterStr = "";
    }
    document.getElementById("message").innerHTML += "Hello " + counterStr + "<br>";
}

function setFunction() {
    setTimeout(sayHello, 500);
}

function setParams() {
    setTimeout(sayHello, 500, "1");
}

function setClosureWrong() {
    for (var j=1,i=2;i<5;i++,j++){
        setTimeout(function() { sayHello(i); }, 500*j);
    }
}

function setClosureRight() {
    for (var j=1,i=6;i<9;i++,j++) {
        setSetTimeout(i,j);
    }
}

function setSetTimeout(counter,delay) {
    setTimeout(function() { sayHello(counter); }, 500*delay);
}

window.onload = function() {
    document.getElementById("functionButton").onclick = setFunction;
    document.getElementById("paramButton").onclick = setParams;
    document.getElementById("closureWrongButton").onclick = setClosureWrong;
    document.getElementById("closureRightButton").onclick = setClosureRight;
};
</script>
</head>
<body>
<h1>setTimeout() Says "Hello"</h1>
<hr>
<form>
    <input type="button" value="As Function" id="functionButton">
    <input type="button" value="With Params" id="paramButton">
    <input type="button" value="With Closure Wrong" id="closureWrongButton">
    <input type="button" value="With Closure Right" id="closureRightButton">
</form>
<br>
<div id="message"></div>
</body>
</html>

```

ONLINE <http://javascriptref.com/3ed/ch12/settimeoutparams.html>

There is some concern about how accurate and fast timers are. A number of developers have noted that when setting timeouts to 0 ms, the effect rate of the timeout can vary by many milliseconds, though it does generally enforce the timed event to be the next action taken. Timing accuracy certainly is not guaranteed, even if order is preserved. It is interesting to note at this edition's writing the inclusion in Gecko-based browsers of an extra parameter that indicates the "lateness" of the timeout in milliseconds. Likely, this is a portend of things to come, and we expect more emphasis on timing details as developers continue to push JavaScript to more time-sensitive tasks.

Window Events

The Window object supports many events. The HTML5 specification attempts to clear up the cross-browser nightmare. Traditionally, most developers stuck with the obviously safe cross-browser window events such as `onblur`, `onerror`, `onfocus`, `onload`, `onunload`, `onresize`, and so on. However, as shown in *Table 12-6*, there are many more events available than those useful few.

Table 12-6 window Events under HTML5

Event	Description
<code>onabort</code>	Invoked generally by the cancellation of an image load, but may happen on any communication that aborts (for example, Ajax calls). Abort events do not have to target the element directly, as any abort event that bubbles through an element can be caught.
<code>onafterprint</code>	Called after a printing event.
<code>onbeforeprint</code>	Called before a print event.
<code>onbeforeunload</code>	Invoked just before a page or object is unloaded from the user-agent.

onblur	Fires when the window loses focus.
oncanplay	Fires when a media element can be played, but not necessarily continuously for its complete duration without potential buffering.
oncanplaythrough	Fires when a media element can be played and should play its complete duration uninterrupted.
onchange	Signals that the form control has lost user focus and its value has been modified during its last access.
onclick	Fires when the user clicks in the window.
oncontextmenu	Fires when the user right-clicks for the context menu.
oncuechange	Fires when the text track of a media item in HTML5 media changes.
ondblclick	Fires when the user double-clicks in the window.
ondrag	Fires when a draggable element is drug around the screen.
ondragend	Occurs at the very end of the drag-and-drop action (should be after ondrag).
ondragenter	Fires when a drug item passes on the element with this event handler—in other words, when the drug item enters into a drop zone.
ondragleave	Fires when a drug item leaves the element with this event handler—in other words, when the drug item leaves a potential drop zone.

ondragover	Fires when an object that is being dragged is over some element with this handler.
ondragstart	Occurs at the very start of a drag-and-drop action.
ondrop	Fires when a drug object is released on some drop zone.
ondurationchange	Fires when the value indicating the duration of a media element changes.
onemptied	Fires when a media element goes into an uninitialized or emptied state, potentially due to some form of a resource reset.
onended	Fires when a media element's playback has ended because the end of the data resource has been reached.
onerror	Used to capture various events generally related to communication using Ajax, though may apply simply to URL reference loads via media elements for including images, audio, video, and others. This attribute is also used for catching script-related errors.
onfocus	Fires when the window gains focus.
onhashchange	Fires when the hash part of the URL changes.
oninput	Fires when input is made to form elements.
oninvalid	Fires when a form field is specified as invalid according to validation rules set via HTML5 attributes such as pattern, min, and max.

onkeydown	Fires when the user presses a key.
onkeypress	Fires when the user presses a key.
onkeyup	Fires when the user releases a key.
onload	Fires when the document is completely loaded into the window. Warning: The timing of this event is not always exact.
onloadeddata	Fires when the user-agent can play back the media data at the current play position for the first time.
onloadedmetadata	Fires when the user-agent has the media's meta data describing the media's characteristics.
onloadstart	Fires when the user-agent begins to fetch media data that may include the initial meta data.
onmessage	Fires when a message hits an element. HTML5 defines a message-passing system between client and server as between documents and scripts that this handler can monitor.
onmousedown	Fires when the user presses a mouse key.
onmousemove	Fires when the user moves the mouse.
onmouseout	Fires when the user moves out of an element.
onmouseover	Fires when the user moves the mouse over an element.
onmouseup	Fires when the user releases a mouse button.
onmousewheel	Fires when the user scrolls the mouse wheel.

onoffline	Fires when user-agent goes offline.
ononline	Fires when user-agent goes back online.
onpause	Fires when a media element pauses by user or script control.
onplay	Fires when a media element starts to play, commonly after a pause has ended.
onplaying	Fires when a media element's playback has just started.
onpagehide	Fires when hiding a page when moving from a history entry.
onpageshow	Fires when showing a page when moving to a history entry.
onpopstate	Fires when the session state changes for the window. This may be due to history navigation or triggered programmatically.
onprogress	Fires when the user-agent is fetching data. Generally applies to media elements, but Ajax syntax has used a similar event.
onratechange	Fires when the playback rate for media changes.
onreadystatechange	Fires whenever the ready state for an object has changed. May move through various states as network fetched data is received.
onredo	Triggered when an action redo is fired.
onreset	Fires when a form in the window is reset.
onresize	Triggered as a user resizes the window.

onscroll	Fires when the window has been scrolled.
onseeked	Fires when the user-agent has just finished the seeking event.
onseeking	Fires when the user-agent is attempting to seek a new media position and has had time to fire the event as the media point of interest has not been reached.
onselect	Fires when text has been selected.
onshow	Fires when a context menu is shown. The event shown remains until the context menu is dismissed.
onstalled	Fires when the user-agent attempts to fetch media data but nothing arrives unexpectedly.
onstorage	Fires when data is committed to the local DOM storage system.
onsubmit	Fires when a form has been submitted.
onsuspend	Fires when a media stream is intentionally not being fetched but is not yet fully loaded.
ontimeupdate	Fires when the time position of the media updates both in standard course of playing or in a seek or jump.
onundo	Fires when an undo is triggered.
onunload	Triggered when the document is unloaded, such as following an outside link or closing the window.
onvolumechange	Fires when the volume attribute or mute attribute value of an HTML5 media tag such as <audio> or <video> changes generally via script or the user's interaction with any shown controls.
onwaiting	Fires when media element play stops but new data is expected shortly.

Adding window events handlers can be set through HTML event attributes on the **<body>** element, like so:

```
<body onload="alert('entering window');" onunload="alert('leaving window');">
```

or more registering events can be set through the Window object:

```
function sayHi() { alert("hi"); }
function sayBye() { alert("bye"); }
// listener style
window.addEventListener("load", sayHi, false);

// direct assignment style
window.onunload = sayBye;
```

Chapter 11 has full details on event handling, in case you are wondering how to bind or test anything.

As time marches on, browser vendors continue to add numerous events to the Window object. A list of those known at this edition's publication is detailed in *Table 12-7*. Check your browser documentation for any others that may have been added since then.

Table 12-7 Selected Proprietary window Events

Event	Description
<code>onactivate</code>	Fires when the object is set as the active element.
<code>onbeforedeactivate</code>	Fires immediately before the active element is changed from one object to another.
<code>onfocusin</code>	Fires just before a window receives focus.
<code>onfocusout</code>	Fires just as the window loses focus, similar to <code>onblur</code> .
<code>onhelp</code>	Fires when the Help key, generally F1, is pressed.
<code>onmozbeforepaint</code>	Fires on the <code>MozBeforePaint</code> event, which is fired when repainting a window through a call to <code>window.mozRequestAnimationFrame()</code> . Likely to be renamed <code>onbeforepaint</code> if it becomes widely used.
<code>onpaint</code>	Fires on paint events for the window.
<code>onresizeend</code>	Fires when the resize process ends—usually when the user has stopped dragging the corner of a window.
<code>onresizestart</code>	Fires when the resize process begins—usually when the user has started dragging the corner of a window.

Interwindow Communication Basics

For applications that have multiple windows launched, it is especially important to understand the basics of communicating among windows. Normally, we access the methods and properties of the primary window using the object instance named simply `window`, or more likely we just omit the reference. However, if we want to access another Window, we need to use the name of that window. For example, given a window named “myWindow,” we could access its document object as `mywindow.document`, and thus we could run any method such as writing to the document:

```
mywindow.document.write("Boom!");
```

or accessing it with standard DOM methods:

```
mywindow.document.getElementById("someElement").innerHTML = "Boom!";
```

The key to communicating between windows is knowing the name of the window and then using that name in place of the generic object reference *window*. Of course, there is the important question of how you reference the main window from a created window. The primary way is using the `window.opener` property that references the Window object that created the current window. The simple example here shows how one window creates another and how each is capable of modifying the other's DOM tree, as well as reading script variables:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Simple Window Communication</title>
<script>
function createWindow() {
  secondwindow = window.open("", "example", "height=300,width=200,scrollbars=yes");
  if (secondwindow != null) {
    var windowHTML = "<!DOCTYPE html><html>";
    windowHTML += "<head><title>Second Window</title>";
    windowHTML += "<script>var aVar = 'Set in spawned window';</scr"+"ipt></head>";
    windowHTML += "<body><h1 align='center'>";
    windowHTML += "Another window!</h1><hr><div align='center'><form action="
      '# ' method='get'>";
    windowHTML += "<input type='button' value='Set main red' onclick="
      "'window.opener.document.bgColor=\"red\";'>";
    windowHTML += "<br><input type='button' value='Show My Variable' onclick="
      "'alert(aVar);'>";
    windowHTML += "<br><input type='button' value='Set Your Variable' onclick="
      "'window.opener.aVar="+ "'Set by spawned window'"+";'>";
    windowHTML += "<br><input type='button' value='CLOSE' onclick='self.close();'>";
    windowHTML += "</form></div></body></html>";

    secondwindow.document.write(windowHTML);
    secondwindow.focus();
  }
}

function setRed() {
  if (window.secondwindow) {
    secondwindow.document.bgColor="red";
    secondwindow.focus();
  }
}
```

```

}

var aVar = "I am a value in the main window";

window.onload = function() {
    document.getElementById("createBtn").onclick = createWindow;
    document.getElementById("changeBtn").onclick = setRed;
    document.getElementById("showBtn").onclick = function () {
        alert(window.aVar);
    };

    document.getElementById("setBtn").onclick = function () {
        if (window.secondwindow)
            secondwindow.aVar = "Set by main window";
        else
            alert("Dependent window not up, please spawn it.")
    };
};
</script>
</head>
<body>
<h1>Simple Window Communication</h1>
<form>
<input type="button" value="New window" id="createBtn">
<input type="button" value="Set other red" id="changeBtn">
<input type="button" value="Show My Variable" id="showBtn">
<input type="button" value="Set Your Variable" id="setBtn">
</form>
</body>
</html>

```

ONLINE <http://javascriptref.com/3ed/ch12/simplewindowcommunication.html>

Now, one limitation of this traditional communication method is that it requires that the communicating windows be spawned by the same origin; thus it is not at all possible to talk to windows from other domains. HTML5 introduces new facilities that should allow for a much more flexible message passing system.

Interwindow Message Passing with `postMessage()`

HTML5 expands on the idea of passing data between windows with the `postMessage()` method. The syntax of this method is

```
postMessage(message, targetOrigin)
```

where

- *message* is the message to pass.

- *targetOrigin* is the domain to which the target window must belong.

While you can use wildcards such as “*” to allow any origin, this is not recommended.

Next, you can listen for incoming messages in windows by setting up a handler for `window.onmessage`. The event object sent to the event handling function will contain a *data*, *origin*, and *source* property where *data* is the actual message received, *origin* is the domain the message came from, and *source* is a reference to the Window object that sent the message.

Once again, as there are security concerns communicating between domains, we should check the origin and source carefully. A simple example passing data between two domains held by one of the authors is shown here. The first page is the page sending the message and the second page is the page receiving the message and replying:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>postMessage</title>
</head>
<body>
<h1>postMessage</h1>
<form>
  <input type="button" id="createBtn" value="Open Window"><br>
  <input type="text" value="10" id="number">
  <input type="button" value="Calculate Factorial" id="calculateBtn"><br>
</form>
<div id="message"></div>
<script>
var myWindow;
function createWindow() {
  myWindow = open("http://htmlref.com/examples/childMessage.
html", "mywin", "height=
  300,width=400,scrollbars=yes");
}
function sendMessage() {
```

```

    var number = document.getElementById("number").value;
    myWindow.postMessage(number, "http://htmlref.com");
}
function receiveMessage(event) {
    if (event.origin != "http://htmlref.com") return;
    document.getElementById("message").innerHTML =
        "Message from : " + event.origin + " with a result = " + event.data;
}
window.onload = function() {
    document.getElementById("createBtn").onclick = createWindow;
    document.getElementById("calculateBtn").onclick = sendMessage;
    window.onmessage = receiveMessage;
};
</script>
</body>
</html>

```

ONLINE <http://javascriptref.com/3ed/ch12/postMessageCrossDomain.html>

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>postMessage</title>
</head>
<body>
<h1>postMessage</h1>
<div id="message"></div>
<script>
window.onmessage = function(event) {
    if (event.origin != "http://javascriptref.com") return;
    var number = parseInt(event.data);
    var product = number;
    for (var i=number-1;i>0;i--) {
        product = product * i;
    }
    event.source.postMessage(product, "http://javascriptref.com");
    document.getElementById("message").innerHTML = "Calculated factorial on " +
        event.data + ". The result is " + product;
};
</script>
</body>
</html>

```

ONLINE <http://htmlref.com/examples/childMessage.html>

Figure 12-4 shows the parent page receiving calculations from the child.



postMessage

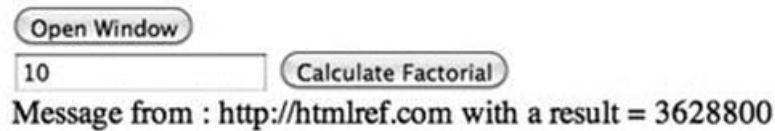


Figure 12-4 Message passing in action

The `postmessage()` passing scheme obviously requires a modern browser, but it suggests a very elegant way to have windows communicate. We wrap up the chapter with a return to the past, while also addressing the relationship between windows and frames.

Frames: A Special Case of Windows

A common misunderstanding among Web developers is the relationship between frames and windows. In reality, both from the perspective of XHTML and JavaScript, each frame shown onscreen is a window that can be manipulated. In fact, when a browser window contains multiple frames, it is possible to access each of the separate Windowobjects through `window.frames[]`, which is an array of the

individual frames in the window. The basic properties useful for manipulating frames are detailed in *Table 12-8*.

Table 12-8 Common WindowProperties Related to Frames

Window Property	Description
frames[]	An array of all the Frame objects contained by the current window.
length	The number of frames in the window. Should be the same value as window.frames.length.
name	The current name of the Window. This is both readable and settable.
parent	A reference to the parent Window.
self	A reference to the current Window.
top	A reference to the top Window. Often the top and the parent will be one in the same unless a <frame> tag loads documents containing more frames.
window	Another reference to the current Window.

The major challenge using frames and JavaScript is to keep the names and relationships between frames clear so that references between frames are formed correctly. Consider you have a document called “frames.html” with the following markup:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>FrameSet Test</title>
</head>
<frameset rows="33%,*,33%">
  <frame src="framereationship.html" name="frame1" id="frame1">
  <frame src="moreframes.html" name="frame2" id="frame2">
  <frame src="framereationship.html" name="frame5" id="frame5">
</frameset>
</html>
```

NOTE Notice that the DOCTYPE statement here is different: HTML5 does not support traditional frames, just inline frames. Where required, we use the HTML4 frameset DOCTYPE for clean validation.

In this case, the window containing this document is considered the parent of the three frames (frame1, frame2, and frame5). While you might expect to use a value such as

```
window.frames.length
```

you probably will actually run the script from within a child frame to determine the number of frames in the window. Thus, you would actually use

```
window.parent.frames.length
```

or just

```
parent.frames.length
```

The parent property allows a window to determine the parent window. We could also use the top property that provides us a handle to the top window that contains all others. This would be written top.frames.length. You do need to be careful, though, because unless you have nested frames, the parent and top may actually be one and the same. In addition, it is possible to access the hosting Frame object with the window.frameElement property.

NOTE Firefox also offers the content property. This returns the topmost window. As it is only supported in Firefox, top is the recommended property.

To access a particular frame, we can use both its name as well as its position in the array, so the following would print out the name of the first frame, which in our case is “frame1”:

```
parent.frames[0].name
```

We could also access the frame from another child frame using parent.frame1, or even parent.frames[“frame1”], using the associate array aspect of an object collection. Remember that a frame contains a

window, so once you have this you can then use all of the Window and Document methods on what the frame contains.

The next example shows the idea of frame names and the way they are related to each other. There are three files that are required for this example—two framesets (frames.html and moreframes.html) and a document (framerelationship.html) that contains a script that prints out the self, parent, and top relationships of frames.

The first frameset file, frames.html, is listed here:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>FrameSet Test</title>
</head>
<frameset rows="33%,*,33%">
    <frame src="framerelationship.html" name="frame1" id="frame1">
    <frame src="moreframes.html" name="frame2" id="frame2">
    <frame src="framerelationship.html" name="frame5" id="frame5">
</frameset>
</html>
```

The second frameset file, moreframes.html, is listed here:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content=
"text/html; charset=utf-8"> <meta <title>More Frames</title>
</head>
<frameset cols="50%,50%">
    <frame src="framerelationship.html" name="frame3" id="frame3">
    <frame src="framerelationship.html" name="frame4" id="frame4">
</frameset>
</html>
```

The document, framerelationship.html, is listed here:

```

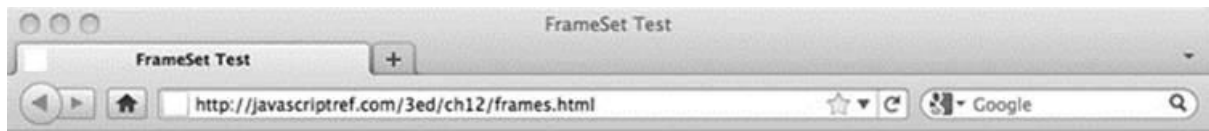
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Frame Relationship Viewer</title>
</head>
<body>
<script>
  var msg="";
  var i = 0;
  msg += "<h2>Window: " + window.name + "</h2><hr>";
  if (self.frames.length > 0)
  {
    msg += "self.frames.length = " + self.frames.length + "<br>"
    for (i=0; i < self.frames.length; i++)
      msg += "self.frames["+i+"].name = " + self.frames[i].name + "<br>";
  }
  else
    msg += "Current window has no frames directly within it<br>";
  msg+="<br>";
  if (parent.frames.length > 0)
  {
    msg += "parent.frames.length = " + parent.frames.length + "<br>"
    for (i=0; i < parent.frames.length; i++)
      msg += "parent.frames["+i+"].name = " + parent.frames[i].name +
"<br>";
  }
  msg+="<br>";
  if (top.frames.length > 0) {
    msg += "top.frames.length = " + top.frames.length + "<br>"
    for (i=0; i < top.frames.length; i++)
      msg += "top.frames["+i+"].name = " + top.frames[i].name + "<br>";
  }

  document.write(msg);
</script>
</body>
</html>

```

ONLINE <http://javascriptref.com/3ed/ch12/frames.html>

The relationships using these example files are shown in *Figure 12-5*.



Window: frame1

Current window has no frames directly within it

```
parent.frames.length = 3  
parent.frames[0].name = frame1  
parent.frames[1].name = frame2  
parent.frames[2].name = frame5
```

```
top.frames.length = 3  
top.frames[0].name = frame1  
top.frames[1].name = frame2  
top.frames[2].name = frame5
```

Window: frame3

Current window has no frames directly within it

```
parent.frames.length = 2  
parent.frames[0].name = frame3  
parent.frames[1].name = frame4
```

```
top.frames.length = 3  
top.frames[0].name = frame1  
top.frames[1].name = frame2  
top.frames[2].name = frame5
```

Window: frame4

Current window has no frames directly within it

```
parent.frames.length = 2  
parent.frames[0].name = frame3  
parent.frames[1].name = frame4
```

```
top.frames.length = 3  
top.frames[0].name = frame1  
top.frames[1].name = frame2  
top.frames[2].name = frame5
```

Window: frame5

Current window has no frames directly within it

```
parent.frames.length = 3  
parent.frames[0].name = frame1  
parent.frames[1].name = frame2  
parent.frames[2].name = frame5
```

```
top.frames.length = 3  
top.frames[0].name = frame1  
top.frames[1].name = frame2  
top.frames[2].name = frame5
```

Figure 12-5 Frame relationships

Once you understand the relationships between frames, you will find it much easier to assign variables to particular frames within deeper pages rather than using the `parent.frames[]` array all the time. For example, given a simple frameset such as this:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd"> <html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Two Frames</title>
</head>
<frameset cols="300,* ">
    <frame src="navigation.html" name="frame1" id="frame1">
    <frame src="content.html" name="frame2" id="frame2">
</frameset>
</html>

```

you might set a variable to reference the content frame within the navigation Window, like so:

```

var contentFrame = parent.frames[1]; // or reference by name

```

This way, you could just reference things by *contentFrame* rather than the long array path.

Inline Frames

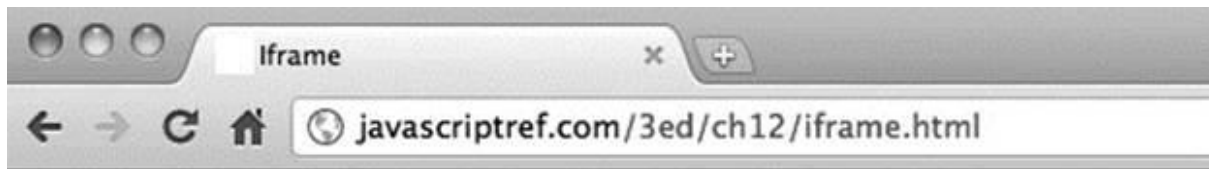
One variation of frames that deserves special attention is the `<iframe>`, or inline frame, because it is preserved under HTML5. The idea with an inline frame is that you can add a frame directly into a document without using a frameset. For example, this example

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Iframe</title>
</head>
<body>
<h1>Regular Content here</h1>
<iframe src="http://www.google.com" name="iframe1" id="iframe1" height=
"200" width="400"></iframe>
<h1>More content here</h1>
</body>
</html>

```

produces a page something like this:



Regular Content here



More content here

ONLINE <http://javascriptref.com/3ed/ch12/iframe.html>

This then begs the question, “How do we control this type of frame?” In reality, it is much easier since it is within the `frames[]` array of the current window. Furthermore, if named, you can use DOM methods such as `getElementById()` to access the object. The simple example here demonstrates this idea:

```
<iframe src="http://www.google.com" name="iframe1" id="iframe1"
height="200" width="200"></iframe>
<form>
<input type="button" value="Load by Frames Array" onclick=
"frames['iframe1'].location='http://www.javascriptref.com';">
<input type="button" value="Load by DOM" onclick=
"document.getElementById('iframe1').src='http://www.pint.com';">
</form>
```


While inline frames seem to be a simplification of standard frames, they are far more interesting than these examples suggest. In fact, we'll see in *Chapter 15* that `<iframe>` tags serve as a non-Ajax method for JavaScript to communicate with a Web server. For now, though, we put off this advanced application and study some more common JavaScript frame applications.

Applied Frames

Now that we are familiar with frame naming conventions, it is time to do something with them. In this section, we present some solutions for common frame problems and hint at the larger issues with frame usage.

Loading Frames

A common question developers have with HTML is how to load multiple frames with a link. XHTML provides the `target` attribute to target a single frame, such as *framename*, like so:

```
<a href="http://www.google.com" target="framename">Google</a>
```

However, how would you target two or more frames with a single link click? The answer, of course, is by using JavaScript. Consider the frameset here:

```
<frameset cols="300,* ">
  <frame src="navigation.html" name="frame1" id="frame1">
  <frame src="content.html" name="frame2" id="frame2">
  <frame src="morecontent.html" name="frame3" id="frame3">
</frameset>
```

In this case, we want a link in the `navigation.html` file to load two windows at once. We could write a simple set of JavaScript statements to do this, like so:

```
<a href="javascript: parent.frames['frame2'].location='http://www.google.com';
parent.frames['frame3'].location='http://www.javascriptref.com';">Two Sites</a>
```

This approach can get somewhat unwieldy, so you might instead want to write a function called *loadFrames()* to do the work. You might even consider using a generic function that takes two arrays—one

with frames and one with URL targets—and loads each one by one, as demonstrated here:

```
<script>
function loadFrames(theFrames,theURLs) {
  if ( (loadFrames.arguments.length != 2) || (theFrames.length != theURLs.length) )
    return;
  for (var i=0;i<theFrames.length;i++)
    theFrames[i].location = theURLs[i];
}
</script>
<a href="javascript:loadFrames([parent.frames['frame2'],
parent.frames['frame3'],parent.frames['frame4']],
['http://www.google.com','http://www.javascriptref.com',
'http://www.ucsd.edu']);">Three Sites</a>
```

Frame Busting

While frames can be very useful for building some complex user interfaces and comparing documents, they also can cause Web designers significant problems. For example, some sites will put frames around all outbound links and “capture” the browsing session. Often, site designers will employ a technique called “frame busting” to destroy any enclosing frameset their page may be enclosed within. This is very easy using the following script that sets the topmost frame’s current location to the value of the page that should not be framed:

```
function frameBuster() {
  if (window != top)
    top.location.href = location.href;
}
window.onload = frameBuster;
```

Frame Building

The converse problem to the one solved by frame busting would be to avoid having framed windows displayed outside of their framing context. This occasionally happens when users bookmark a piece of a frameset or launch a link from a frameset into a new window. The basic idea would be to have all framed documents make sure they are inside of frames by looking at each window’s location object, and if they are not, to dynamically rebuild the frameset document. For

example, given a simple two-frame layout such as in a file called frameset.html:

```
<frameset cols="250,*">
  <frame src="navigation.html" name="navigation" id="navigation">
  <frame src="content.html" name="content" id="content">
</frameset>
```

You might be worried that a user could bookmark or directly enter the navigation.html or content.html URL. To rebuild the frameset in navigation.html and content.html, you might have

```
<script>
if (parent.location.href == self.location.href)
  window.location.href = "frameset.html";
</script>
```

which would detect if the page was outside its frameset and rebuild it. Of course, this is a very simplistic example, but it gives the basic idea of frame building. The script can be expanded and a variety of tricks employed to preserve the state of the navigation and content pages.

All the efforts made in the last few sections reveal that frames really do have their downsides. While they may provide for stable user interfaces, they are not terribly bookmarking friendly, more than occasionally have printing problems, and not well handled by search engines. As we demonstrated, you can certainly use JavaScript to solve the problems with frames, but it might be better simply to avoid using them in many cases. Before concluding our discussion of frames, let's take a final look at interwindow communication for state management using frames and JavaScript.

State Management with Frames

One aspect of frames that some developers found useful early on with JavaScript was the ability to save variable state across multiple page views. As we saw with windows previously, it is possible to access the variable space of one window from another Window, and the same holds for frames. Using a special type of frameset where a small frame that is hard for a user to notice is used, we can create a space to

hold variables across page loads. Consider for example, the frameset in the file `stateframes.html` shown here:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
<meta charset="utf-8">
<title>State Preserve Frameset</title>
</head>
<frameset rows="99%,*" >
    <frame src="mainframe.html" name="frame1" id="frame1" frameborder="0">
    <frame src="stateframe.html" name=
        "stateframe" id="stateframe" frameborder="0" scrolling="no"
        noresize="noresize">
</frameset>
</html>
```

In this case, we have a very small frame called *stateframe* that will be used to save variables across page loads. The contents of `stateframe.html`, `mainframe.html`, and `mainframe2.html` are shown here. Notice how, by referencing the parent frame, we are able to access the hidden frame's variable *username* on any page.

The `stateframe.html` file is shown here:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Variables</title>
</head>
<body>
<script>
    var username;
</script>
</body>
</html>
```

The `mainframe.html` file is shown here:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>State Preserve 1</title>
<script>
function saveValue() {
    parent.stateframe.username = document.getElementById("username").value;
}
window.onload = function() {
    document.getElementById("saveButton").onclick = saveValue;
}
</script>
</head>
<body>
<h1>JS State Preserve</h1>
<form>
    <input type="text" id="username" value="" size="30" maxlength="60">
    <input type="button" value="Save Value" id="saveButton">
</form>
<div style="align:center;">
    <a href="mainframe2.html">Next page</a>
</div>
</body>
</html>

```

The mainframe2.html file is shown here:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>State Preserve 2</title>
</head>
<body>
<script>
if (!(parent.stateframe.username) || (parent.stateframe.username == ""))
    document.write("<h1 style='align:center;'>Sorry we haven't met before</h1>");
else
    document.write("<h1 style='align:center;'>Welcome to the page "+parent.stateframe.username+"!</h1>");
</script>
<div style="text-align:center;">
    <a href="mainframe.html">Back to previous page</a>
</div>
</body>
</html>

```

ONLINE <http://javascriptref.com/3ed/ch12/stateframes.html>

Obviously, as compared to pushstate() methods and other more modern features, the use of simple interwindow communications with

frames to maintain state is a bit primitive. However, we'll see that in nearly any case, the security implications of all of these client-side state preservation mechanisms leaves a bit to be desired. Given the hostile nature of the Internet, programmers are strongly encouraged to rely on traditional state management mechanisms such as cookies to maintain state between pages in a site. More information on state management can be found in *Chapter 16*.

PHP programs

1) Fibonacci Series

Fibonacci series is the one in which you will get your next term by adding previous two numbers.

For example,

1. 0 1 1 2 3 5 8 13 21 34
2. Here, $0 + 1 = 1$
3. $1 + 1 = 2$
4. $3 + 2 = 5$

and so on.

Logic:

- Initializing first and second number as 0 and 1.
- Print first and second number.
- From next number, start your loop. So third number will be the sum of the first two numbers.

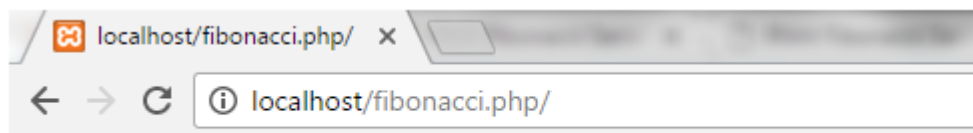
Example:

We'll show an example to print the first 12 numbers of a Fibonacci series.

1. `<?php`
2. `$num = 0;`
3. `$n1 = 0;`

```
4. $n2 = 1;
5. echo "<h3>Fibonacci series for first 12 numbers: </h3>";
6. echo "\n";
7. echo $n1.' '.$n2.' ';
8. while ($num < 10 )
9. {
10.  $n3 = $n2 + $n1;
11.  echo $n3.' ';
12.  $n1 = $n2;
13.  $n2 = $n3;
14.  $num = $num + 1;
15. ?>
```

Output:



Fibonacci series for first 12 numbers:

0 1 1 2 3 5 8 13 21 34 55 89

2) Leap Year Program

A leap year is the one which has 366 days in a year. A leap year comes after every four years. Hence a leap year is always a multiple of four.

For example, 2016, 2020, 2024, etc are leap years.

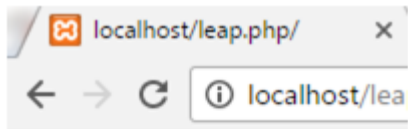
Leap Year Program

This program states whether a year is leap year or not from the specified range of years (1991 - 2016).

Example:

```
1. <?php
2. function isLeap($year)
3. {
4.     return (date('L', mktime(0, 0, 0, 1, 1, $year))==1);
5. }
6. //For testing
7. for($year=1991; $year<2016; $year++)
8. {
9.     If (isLeap($year))
10.    {
11.        echo "$year : LEAP YEAR<br />\n";
12.    }
13.    else
14.    {
15.        echo "$year : Not leap year<br />\n";
16.    }
17.}
18.??>
```

Output:



1991 : Not leap year
1992 : LEAP YEAR
1993 : Not leap year
1994 : Not leap year
1995 : Not leap year
1996 : LEAP YEAR
1997 : Not leap year
1998 : Not leap year
1999 : Not leap year
2000 : LEAP YEAR
2001 : Not leap year
2002 : Not leap year
2003 : Not leap year
2004 : LEAP YEAR
2005 : Not leap year
2006 : Not leap year
2007 : Not leap year
2008 : LEAP YEAR
2009 : Not leap year
2010 : Not leap year
2011 : Not leap year
2012 : LEAP YEAR
2013 : Not leap year
2014 : Not leap year
2015 : Not leap year

3) Factorial Program

The factorial of a number n is defined by the product of all the digits from 1 to n (including 1 and n).

For example,

1. $4! = 4*3*2*1 = 24$
2. $6! = 6*5*4*3*2*1 = 720$

Note:

- It is denoted by $n!$ and is calculated only for positive integers.
- Factorial of 0 is always 1.

The simplest way to find the factorial of a number is by using a loop.

There are two ways to find factorial in PHP:

- Using loop
- Using recursive method

Logic:

- Take a number.
- Take the descending positive integers.
- Multiply them.

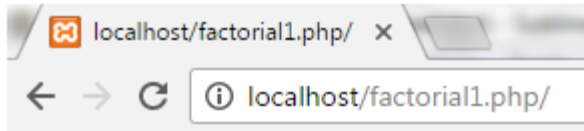
Factorial in PHP

Factorial of 4 using for loop is shown below.

Example:

1. `<?php`
2. `$num = 4;`
3. `$factorial = 1;`
4. `for ($x=$num; $x>=1; $x--)`
5. `{`
6. `$factorial = $factorial * $x;`
7. `}`
8. `echo "Factorial of $num is $factorial";`
9. `?>`

Output:



Factorial of 4 is 24

4) Armstrong Number

An Armstrong number is the one whose value is equal to the sum of the cubes of its digits.

0, 1, 153, 371, 407, 471, etc are Armstrong numbers.

For example,

1. $407 = (4*4*4) + (0*0*0) + (7*7*7)$
2. $= 64 + 0 + 343$
3. $407 = 407$

Logic:

- Take the number.
- Store it in a variable.
- Take a variable for sum.
- Divide the number with 10 until quotient is 0.
- Cube the remainder.
- Compare sum variable and number variable.

Armstrong number in PHP

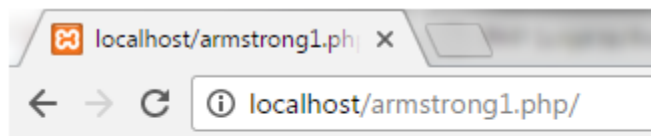
Below program checks whether 407 is Armstrong or not.

Example:

1. `<?php`
2. `$num=407;`
3. `$total=0;`

```
4. $x=$num;
5. while($x!=0)
6. {
7. $rem=$x%10;
8. $total=$total+$rem*$rem*$rem;
9. $x=$x/10;
10.}
11.if($num==$total)
12.{
13.echo "Yes it is an Armstrong number";
14.}
15.else
16.{
17.echo "No it is not an armstrong number";
18.}
19. ?>
```

Output:



Yes it is an Armstrong number

5) Palindrome Number

A palindrome number is a number which remains same when its digits are reversed.

For example, number 24142 is a palindrome number. On reversing it we'll get the same number.

Logic:

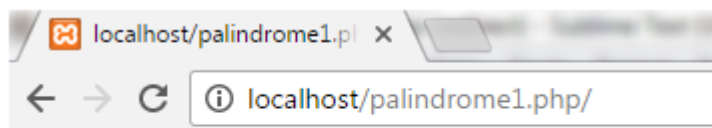
- Take a number.
- Reverse the input number.
- Compare the two numbers.
- If equal, it means number is palindrome

Palindrome Number in PHP

Example:

```
1. <?php
2. function palindrome($n){
3. $number = $n;
4. $sum = 0;
5. while(floor($number)) {
6. $rem = $number % 10;
7. $sum = $sum * 10 + $rem;
8. $number = $number/10;
9. }
10. return $sum;
11.}
12. $input = 1235321;
13. $num = palindrome($input);
14. if($input==$num){
15. echo "$input is a Palindrome number";
16.} else {
17. echo "$input is not a Palindrome";
18.}
19. ?>
```

Output:



1235321 is a Palindrome number


User Interface Elements



When designing your interface, try to be consistent and predictable in your choice of interface elements. Whether they are aware of it or not, users have become familiar with elements acting in a certain way, so choosing to adopt those elements when appropriate will help with task completion, efficiency, and satisfaction.

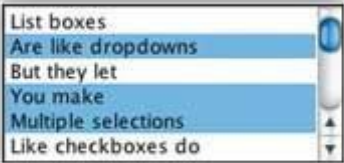


Interface elements include but are not limited to:




- **Input Controls:** checkboxes, radio buttons, dropdown lists, list boxes, buttons, toggles, text fields, date field
- **Navigational Components:** breadcrumb, slider, search field, pagination, slider, tags, icons
- **Informational Components:** tooltips, icons, progress bar, notifications, message boxes, modal windows
- **Containers:** accordion

Input Controls



Element	Description	Examples
Checkboxes	Checkboxes allow the user to select one or more options from a set. It is usually best to present checkboxes in a vertical list. More than one column is acceptable as well if the list is long enough that it might require scrolling or if comparison of	



Element	Description	Examples
	terms might be necessary.	
Radio buttons	Radio buttons are used to allow users to select one item at a time.	 An example of radio buttons showing two options: 'Yes' and 'No'. Each option has a small circle to its left, representing the radio button.
Dropdown lists	Dropdown lists allow users to select one item at a time, similarly to radio buttons, but are more compact allowing you to save space. Consider adding text to the field, such as 'Select one' to help the user recognize the necessary action.	 An example of a dropdown list with the text 'Find your state or...' followed by a blue downward arrow icon and a 'Go' button.

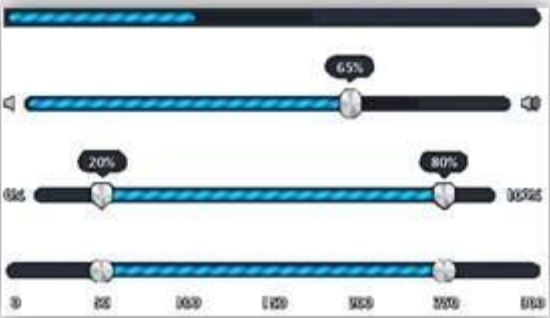


Element	Description	Examples
<p>List boxes</p>	<p>List boxes, like checkboxes, allow users to select a multiple items at a time, but are more compact and can support a longer list of options if needed.</p>	 <p>A screenshot of a list box titled "List boxes". It contains five items: "Are like dropdowns", "But they let", "You make", "Multiple selections", and "Like checkboxes do". The first three items are highlighted in blue, indicating they are selected. A vertical scrollbar is visible on the right side of the list.</p>
<p>Buttons</p>	<p>A button indicates an action upon touch and is typically labeled using text, an icon, or both.</p>	 <p>A screenshot showing three buttons in a row. The first button has an envelope icon and the text "Send". The second button has a Facebook "f" logo and the text "Post". The third button has a Twitter bird logo and the text "Tweet".</p>
<p>Dropdown Button</p>	<p>The dropdown button consists of a button that when clicked displays a drop-down list of mutually exclusive items.</p>	 <p>A screenshot of a dropdown button. The button itself is a gear icon with a downward arrow. The dropdown menu is open, showing three options: "General Settings", "Your Profile" (which is highlighted in blue), and "Sign Out".</p>

Element	Description	Examples
<p>Toggles</p>	<p>A toggle button allows the user to change a setting between two states. They are most effective when the on/off states are visually distinct.</p>	
<p>Text fields</p>	<p>Text fields allow users to enter text. It can allow either a single line or multiple lines of text.</p>	
<p>Date and time pickers</p>	<p>A date picker allows users to select a date and/or time. By using the picker, the information is consistently formatted and input into the system.</p>	


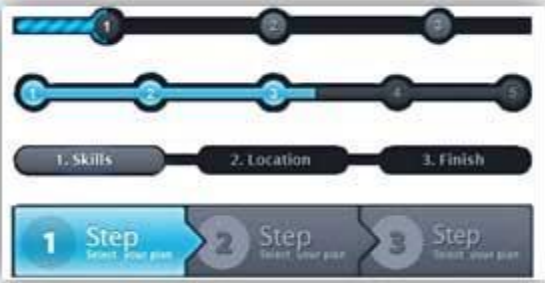
Navigational Components



Element	Description	Examples
<p>Search Field</p>	<p>A search box allows users to enter a keyword or phrase (query) and submit it to search the index with the intention of getting back the most relevant results. Typically search fields are single-line text boxes and are often accompanied by a search button.</p>	 <p>The image shows three different search field designs. The top one is a dropdown menu titled 'Navigation 1' with a search icon, listing options like 'Navigation Crystal Clear', 'Navigation Menu', 'Navigation Soft Style', 'Navigation Class', and 'Navigation Plastic'. The middle one is a search bar with the text 'Enter Keywords', a search icon, and three radio buttons labeled 'Search Option One', 'Search Option Two', and 'Search Option Three'. The bottom one is a search bar with 'Enter Keywords', a 'Category' dropdown menu, and a 'SEARCH' button. The dropdown menu is open, showing options: 'Everything', 'Entries', 'Photos', 'Videos', and 'Audio'.</p>
<p>Breadcrumb</p>	<p>Breadcrumbs allow users to identify their current location within the system by providing a clickable trail of proceeding pages to navigate by.</p>	 <p>The image shows a horizontal breadcrumb trail with the text: Home > Folder Index Page > Page You're On. The text is in a light blue color and is contained within a white rectangular box with a thin border.</p>


Element	Description	Examples
<p>Pagination</p>	<p>Pagination divides content up between pages, and allows users to skip between pages or go in order through the content.</p>	
<p>Tags</p>	<p>Tags allow users to find content in the same category. Some tagging systems also allow users to apply their own tags to content by entering them into the system.</p>	

Element	Description	Examples
<p>Sliders</p>	<p>A slider, also known as a track bar, allows users to set or adjust a value. When the user changes the value, it does not change the format of the interface or other info on the screen.</p>	 <p>The image shows four different slider controls. The top one is a simple horizontal bar with a blue gradient and a white knob. The second one has a blue gradient bar, a white knob, and a '65%' label above it. The third one has a blue gradient bar, a white knob, and '20%' and '80%' labels above it. The bottom one has a black bar with a white knob and a scale from 0 to 1000 below it.</p>
<p>Icons</p>	<p>An icon is a simplified image serving as an intuitive symbol that is used to help users to navigate the system. Typically, icons are hyperlinked.</p>	 <p>The image shows a 3x4 grid of 12 icons. The icons include a calendar, a document, a camera, a laptop, a folder, a file, a person, a group of people, a list, a gear, a calendar, and a window.</p>
<p>Image Carousel</p>	<p>Image carousels allow users to browse through a set of items and make a selection of one if they so choose. Typically, the images are hyperlinked.</p>	 <p>The image shows a horizontal carousel with five gray rectangular items. It has a left arrow on the far left and a right arrow on the far right. Below the items is a row of five small circles, with the second one from the left being filled, indicating the current position.</p>

Information Components

Element	Description	Examples
<p>Notifications</p>	<p>A notification is an update message that announces something new for the user to see. Notifications are typically used to indicate items such as, the successful completion of a task, or an error or warning message.</p>	
<p>Progress Bars</p>	<p>A progress bar indicates where a user is as they advance through a series of steps in a process.</p>	

Element	Description	Examples
	<p>Typically, progress bars are not clickable.</p>	
<p>Tool Tips</p>	<p>A tooltip allows a user to see hints when they hover over an item indicating the name or purpose of the item.</p>	 <p>The image shows two examples of tooltips. The first is a short, dark blue tooltip with a white border and a white arrow pointing down, containing the text "Tooltip under the text." The second is a taller, dark blue tooltip with a white border and a white arrow pointing down, containing three lines of text: "Here is the sample of tall tooltip that contains three lines or more." followed by "More." in a smaller font.</p>
<p>Message Boxes</p>	<p>A message box is a small window that provides information to users and requires them to take an action before they can move forward.</p>	 <p>The image shows a message box with a light gray background and a white border. It has a title bar at the top that says "This is a box". Below the title bar is a block of text: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce metus. Pellentesque sit amet velit. Phasellus non quam. Nulla diam purus, tristique quis, eleifend ac, molestie eu, sapien. Vestibulum ante ipsum primis. Vestibulum ante ipsum primis." At the bottom right of the message box is a blue button with the text "READ MORE" in white.</p>

Element	Description	Examples
Modal Window (pop-up)	A modal window requires users to interact with it in some way before they can return to the system.	

Containers

Element	Description	Examples
Accordion	An accordion is a vertically stacked list of items that utilizes show/ hide functionality. When a label is clicked, it expands the section showing the content within. There can have one or more items showing at a time and may have default states that reveal one or more sections without the user clicking	

Web Application Security

Web application security refers to a variety of processes, technologies, or methods for protecting web servers, web applications, and web services such as APIs from attack by Internet-based threats. Web application security is crucial to protecting data, customers, and organizations from data theft, interruptions in business continuity, or other harmful results of cybercrime.

What Is Web Application Security?

By most estimates, more than three-quarters of all cybercrime targets applications and their vulnerabilities. Web application security products and policies strive to protect applications through measures such as web application firewalls (WAFs), multi-factor authentication (MFA) for users, the use, protection, and validation of cookies to maintain user state and privacy status, and various methods for validating user input to ensure it is not malicious before that input is processed by an application.

Why Is Web Application Security Important?

The world today runs on apps, from online banking and remote work apps to personal entertainment delivery and e-commerce. It's no wonder that applications are a primary target for attackers, who exploit vulnerabilities such as design flaws as well as weaknesses in APIs, open-source code, third-party widgets, and access control.

Common attacks against web applications include:

- Brute force
- Credential stuffing
- [SQL injection](#) and formjacking injections
- [Cross-site scripting](#)
- [Cookie poisoning](#)
- Man-in-the-middle (MITM) and man-in-the-browser attacks
- Sensitive data disclosure
- Insecure deserialization
- Session hijacking

One recent study¹ estimated that cybercrime will cost \$5.2 trillion in lost value across all industries by 2024. Another estimated the losses will reach \$6 trillion annually before then². Security devices and technologies are crucial for limiting, if not eliminating, such costs. In addition to direct financial and data theft, web application threats can destroy assets, customer goodwill, and business reputations. That makes web application security imperative for organizations of all sizes.

How Does Web Application Security Work?

Different approaches to web application security address different vulnerabilities. [Web application firewalls \(WAFs\)](#), among the more comprehensive, defend against many types of attack by monitoring and filtering traffic between the web application and any user. Configured with policies that help determine what traffic is safe and what isn't, a WAF can block malicious traffic, preventing it from reaching the web application and preventing the app from releasing any unauthorized data.

Other web application security methods focus on user authentication and access management, app vulnerability scanners, cookie management, traffic visibility, and IP denylists, for instance..

Common web app vulnerabilities

the top 10 most common application vulnerabilities include:

- **Injection.** An injection happens when a bad actor sends invalid data to the web app to make it operate differently from the intended purpose of the application.
- **Broken Authentication.** A broken authentication vulnerability allows a bad actor to gain control over an account within a system or the entire system.
- **Sensitive Data Exposure.** Sensitive data exposure means data is vulnerable to being exploited by a bad actor when it should have been protected.
- **XML External Entities (XXE).** A type of attack against an application that parses XML input and occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser.
- **Broken Access Control.** When components of a web application are accessible instead of being protected like they should be, leaving them vulnerable to data breaches.
- **Security Misconfigurations.** Incorrectly misconfiguring a web application provides bad actors with an easy way in to exploit sensitive information.
- **Cross Site Scripting (XSS).** An XSS attack means a bad actor injects malicious client-side scripts into a web application.

- **Insecure Deserialization.** Bad actors will exploit anything that interacts with a web application—from URLs to serialized objects—to gain access.
- **Using Components with Known Vulnerabilities.** Instances such as missed software and update change logs can serve as big tip-offs for bad actors looking for ins into a web application. Disregarding updates can allow a known vulnerability to survive within a system.
- **Insufficient Logging and Monitoring.** Lack of efficient logging and monitoring processes increases the chances of a web app being compromised



Tutorial 13

Working with Windows and Frames

Enhancing a Web Site with Interactive Windows



Objectives

- Learn about the properties of the window object
- Create permanent and transient status bar messages
- Work with the properties of the location and history objects
- Apply automatic page navigation to a Web site



Objectives

- Use JavaScript to create a pop-up window
- Learn how to adjust your code to accommodate pop-up blockers
- Work with the properties and methods of pop-up windows
- Create alert, confirm, and prompt dialog boxes



Objectives

- Understand how to write content directly into a pop-up window
- Study how to work with modal and modeless windows



Objectives

- Work with frame and frameset objects
- Study how to navigate between frames
- Learn how to change the content of a frame
- Study how to change a frame layout
- Block frames and force pages into framesets
- Learn how to work with inline frames



Working with the Window Object

- JavaScript considers the browser window an object, which it calls the **window object**

Property	Description
closed	Returns a Boolean value indicating whether the window has been closed
defaultStatus	Defines the default message displayed in the status bar
document	The document object displayed in the window
frames	The collection of frames within the window
history	The history object, containing a list of Web sites visited within that window
innerHeight	The inner height of the window excluding all toolbars, scrollbars, and other features (Netscape only)
innerWidth	The inner width of the window excluding all toolbars, scrollbars, and other features (Netscape only)
location	The location object containing the URL of the current Web document
name	The name of the window
opener	The source browser window, which opened the current window
outerHeight	The outer height of the window including all toolbars, scrollbars, and other features (Netscape only)
outerWidth	The outer width of the window including all toolbars, scrollbars, and other features (Netscape only)
scrollbars	The scrollbar object contained in the browser window
status	The temporary or transient message displayed in the status bar
statusbar	The status bar object used for displaying messages in the browser window
toolbar	A Boolean value indicating whether the window's toolbar is visible



Working with the Window Object

- To set a property of the window object
`windowObject.property = "value"`
- In some cases, can leave the window object reference out
`window.innerHeight = "300";`
- If the property is an object, you can drop the reference to the window
`location = "href"`



Working with Status Bars

- The borders of a browser window, including items such as the toolbars and scrollbars, are collectively referred to as the window's **chrome**
- Common to all browsers is the **status bar**



Working with Status Bars

- Setting the Default Status Bar Message
 - The **permanent status bar message** is the message that appears in the status bar by default

```
<title>Teresa Jenner's iMusicHistory course</title>
<link href="main.css" rel="stylesheet" type="text/css" />
</head>
<body onload="window.defaultstatus='welcome to iMusicHistory'">
```

status bar message



Working with Status Bars

- Creating a transient status bar message
 - A **transient status bar message** appears only temporarily in response to an event occurring within the browser

```
windowObject.status="text";  
return true;
```
 - Transient status bar messages remain until some other event replaces them



Working with Status Bars

- Creating a transient status bar message

```
<!-- Begin navigation text cell -->
<tr>
<td valign="middle" bgcolor="#99CC99" align="center" class="nav">
<a class="nav" href="default.htm"
  onmouseover="window.status='Learn more about iMusichistory';return true"
  onmouseout="window.status='';return true">Home</a>&nbsp;&nbsp;&nbsp;-&nbsp;&nbsp;&nbsp;
<a class="nav" href="lesson3.htm">Lessons</a>&nbsp;&nbsp;&nbsp;-&nbsp;&nbsp;&nbsp;
<a class="nav" href="quiz.htm">Quiz</a>&nbsp;&nbsp;&nbsp;-&nbsp;&nbsp;&nbsp;
<a class="nav" href="terms.htm">Glossary</a><br />
</td>
</tr>
```



Working with the History and Location Objects

- The **location object** contains information about the page that is currently displayed in the window
- The **history object** holds a list of the sites the Web browser has displayed before reaching the current page in the window



Working with the History and Location Objects

- Moving Forward and Backward in the History List

```
history.back();
```

```
history.forward();
```

```
windowObject.history.back();
```

- To go to a particular page in the history

```
history.go(integer);
```



Working with the History and Location Objects

- Automatic Page Navigation
 - Two ways to redirect the user
 - One way is to add a command to the <meta> tag
 - The other is to create a JavaScript program that runs when the page is loaded and opens the new page automatically

```
<meta http-equiv = "Refresh" content = "sec"; URL = "url" />
```

```
windowObject.location.href = "url";
```




Working with the History and Location Objects

- Automatic Page Navigation

```
<title>iMusicHistory has moved</title>
<meta http-equiv="Refresh" content="8;URL=default.htm" />

<script type="text/javascript">
function redirect() {
    setTimeout("location.href='default.htm'",8000);
}
</script>

<style>
body {text-align: center; background-color: white}
h1 {color:red; font-family:Arial, Helvetica, sans-serif}
p {font-size:medium}
</style>
</head>

<body onload="redirect()">
<h1>Teresa Jenner's<br />
    Music History Course<br />
    has Moved<hr /></h1>
```



Working with the History and Location Objects

- Security Issues
 - Netscape (version 4.0 and above) uses signed scripts to request permission to access restricted information
 - Signed scripts are not available in Internet Explorer and prior to Netscape version 4, these properties were not available at all from a script



Creating New Browser Windows

- Windows that open in addition to the main browser window are called **secondary windows** or **pop-up windows**



Creating New Browser Windows

- Opening New Windows with HTML
 - If you want one of your links to open the target document in a new window, you specify the window name using the target property
- ```
link
text
```



# Creating New Browser Windows

- Opening New Windows with JavaScript
  - The JavaScript command to create a new browser window is  
`window.open("url", "name", "features")`



# Creating New Browser Windows

- Setting the Features of a Pop-up Window
  - The feature list obeys the following syntax:  
`"feature1=value1, feature2=value2...featureN=valueN"`



# Creating New Browser Windows

- Setting the Features of a Pop-up Window

| Feature       | Description                                                                                          | Value   |
|---------------|------------------------------------------------------------------------------------------------------|---------|
| alwaysLowered | Sets the window below all other windows (Netscape only)                                              | yes/no  |
| alwaysRaised  | Sets the window above all other windows (Netscape only)                                              | yes/no  |
| dependent     | Window is a dependent of the parent window that created it and closes when it closes (Netscape only) | yes/no  |
| fullscreen    | Displays the window in full screen mode (Internet Explorer only)                                     | yes/no  |
| height        | Window height, in pixels                                                                             | integer |
| hotkeys       | Disables keyboard hotkeys in the window (Netscape only)                                              | yes/no  |
| innerHeight   | Inner height of the window, in pixels (Netscape only)                                                | integer |
| innerWidth    | Inner width of the window, in pixels (Netscape only)                                                 | integer |
| left          | Sets the screen coordinate of the left edge of the window, in pixels (Internet Explorer only)        | integer |
| location      | Displays the location bar in the window                                                              | yes/no  |
| menubar       | Displays the menu bar in the window                                                                  | yes/no  |
| outerHeight   | Outer height of the window, in pixels (Netscape only)                                                | integer |
| outerWidth    | Outer width of the window, in pixels (Netscape only)                                                 | integer |
| resizable     | Allows users to resize the window                                                                    | yes/no  |
| screenX       | Sets the screen coordinate of the left edge of the window, in pixels (Netscape only)                 | integer |
| screenY       | Sets the screen coordinate of the top edge of the window, in pixels (Netscape only)                  | integer |
| scrollbars    | Displays scrollbars in the window                                                                    | yes/no  |
| status        | Displays the status bar in the window                                                                | yes/no  |
| top           | Sets the screen coordinate of the top edge of the window, in pixels (Internet Explorer only)         | integer |
| titlebar      | Displays the title bar (Netscape only)                                                               | yes/no  |
| toolbar       | Displays the window's toolbar                                                                        | yes/no  |
| width         | Sets the width of the window, in pixels                                                              | integer |
| z-lock        | Prevents the window from rising above other windows (Netscape only)                                  | yes/no  |



# Creating New Browser Windows

- Working with Pop-up Blockers
  - Pop-up blockers prevent pop-up windows from opening
  - You can have the browser check whether the pop-up window has been opened or not







# Creating New Browser Windows

- Adding a Pop-up Window to the iMusicHistory Site

```
<title>Lesson 3, iMusicHistory</title>
<link href="main.css" rel="stylesheet" type="text/css" />
<script type="text/javascript">
function popwin(url) {
 defwin=window.open(url, "pop", "width=330,height=220,scrollbars=yes");
 testpop=(defwin==null || typeof(defwin)== "undefined") ? true: false;
 return testpop;
}
</script>
</head>
```





# Creating New Browser Windows

- Window Security Issues
  - A browser's ability to open new windows on a user's computer raises some security issues
  - For example, you cannot create a new window with a width and height less than 100 pixels



# Working with Window Methods

- Window Methods

Method	Description
<code>blur()</code>	Removes the focus from the window
<code>close()</code>	Closes the window
<code>focus()</code>	Gives the window the focus
<code>moveBy(dx, dy)</code>	Moves the window <i>dx</i> pixels to the right and <i>dy</i> pixels down
<code>moveTo(x, y)</code>	Moves the top left corner of the window to the screen coordinates ( <i>x</i> , <i>y</i> )
<code>print()</code>	Prints the contents of the window
<code>resizeBy(dx, dy)</code>	Resizes the window by <i>dx</i> pixels to the right and <i>dy</i> pixels down
<code>resizeTo(x, y)</code>	Resizes the window to <i>x</i> pixels wide and <i>y</i> pixels high
<code>scrollBy(dx, dy)</code>	Scrolls the document content in the window by <i>dx</i> pixels to the right and <i>dy</i> pixels down
<code>scrollTo(x, y)</code>	Scrolls the document in the window to the page coordinates ( <i>x</i> , <i>y</i> )



# Working with Window Methods

- The Self and Opener Keywords
  - The **self keyword** refers to the current window
  - Self keyword is synonymous with the window keyword, but you may see it used to improve clarity
  - The **opener keyword** refers to the window or frame that used the `window.open()` method to open the current window



# Creating Dialog Boxes

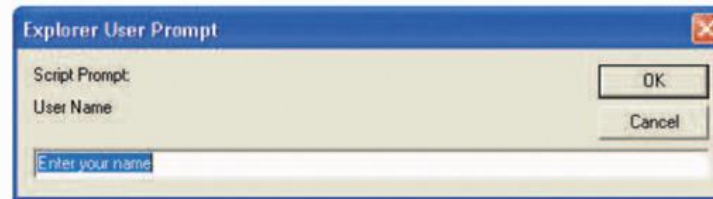
- An **alert dialog box** displays a message along with an OK button
- A **prompt dialog box** displays both a message and a text box in which the user can enter text
- A **confirm dialog box** displays a message along with OK and cancel buttons



# Creating Dialog Boxes



```
alert("Form Completed")
```



```
prompt("User Name", "Enter your name")
```



```
confirm("Continue Program?")
```



# Creating Dialog Boxes

```
<title>iMusicHistory Quiz</title>
<link href="main.css" rel="stylesheet" type="text/css" title="General" />

<script type="text/javascript">
function answer(choice, guess) {
 if (guess) {
 alert(choice + " is correct!");
 } else {
 alert(choice + " is incorrect. Try again");
 }
}
</script>

</head>
```





# Working between Windows

- Writing Content to a Window
    - To write content to a pop-up-window, you use the `document.write()` method
- ```
windowObject.document.write("Content");
```



Working between Windows

- Accessing an Object within a Window
 - Once you specify a window object, you can work with the objects contained in the window's document

```
windowObject.document.getElementById(id) ;
```

```
windowObject.variable
```

```
windowObject.function()
```




Working with Modal and Modeless Windows

- A **modal window** is a window that prevents users from doing work in any other window or dialog box until the window is closed
- A **modeless window** allows users to work in other dialog boxes and windows even if the window stays open

```
windowObject.showModalDialog("url", "arguments",  
    "features")
```

```
windowObject.showModelessDialog("url",  
    "arguments", "features")
```



Working with Modal and Modeless Windows

- Working with the Features List

| Feature | Value | Description |
|--------------|----------------|---|
| dialogHeight | Numeric | Sets the height of the dialog window; the default unit of measurement is "em" in Internet Explorer 4.0, and pixels in Internet Explorer 5.0 |
| dialogLeft | Numeric | Specifies the screen coordinates of the left edge of the dialog box, in pixels |
| dialogTop | Numeric | Specifies the screen coordinates of the top edge of the dialog box, in pixels |
| dialogWidth | Numeric | Sets the width of the dialog window; the default unit of measurement is "em" in Internet Explorer 4.0, and pixels in Internet Explorer 5.0 |
| center | yes, no | Specifies whether to center the dialog box on the desktop |
| dialogHide | yes, no | Specifies whether to hide the dialog box when printing or using print preview |
| edge | sunken, raised | Specifies the edge style of the window; the default is raised |
| help | yes, no | Specifies whether the window displays the context-sensitive Help icon in the title bar |
| resizable | yes, no | Specifies whether the window is resizable |
| scroll | yes, no | Specifies whether the window displays scroll bars |
| status | yes, no | Specifies whether the window displays a status bar |
| unadorned | yes, no | Specifies whether the window displays the browser chrome |



Working with Modal and Modeless Windows

- Exchanging Information between the Windows
 - Neither the `showModalDialog()` nor the `showModelessDialog()` methods allow direct interaction between the calling browser window and the pop-up window
 - If you need to send information, you must include that data in the `arguments` parameter for the method



Working with Frames

- The name attribute of a frame is used when creating links whose targets are designed to appear in specific frames
- To reference a specific frame in your JavaScript code, you need to use the id attribute

```
<frame id="top" name="top" src="home.htm" />
```



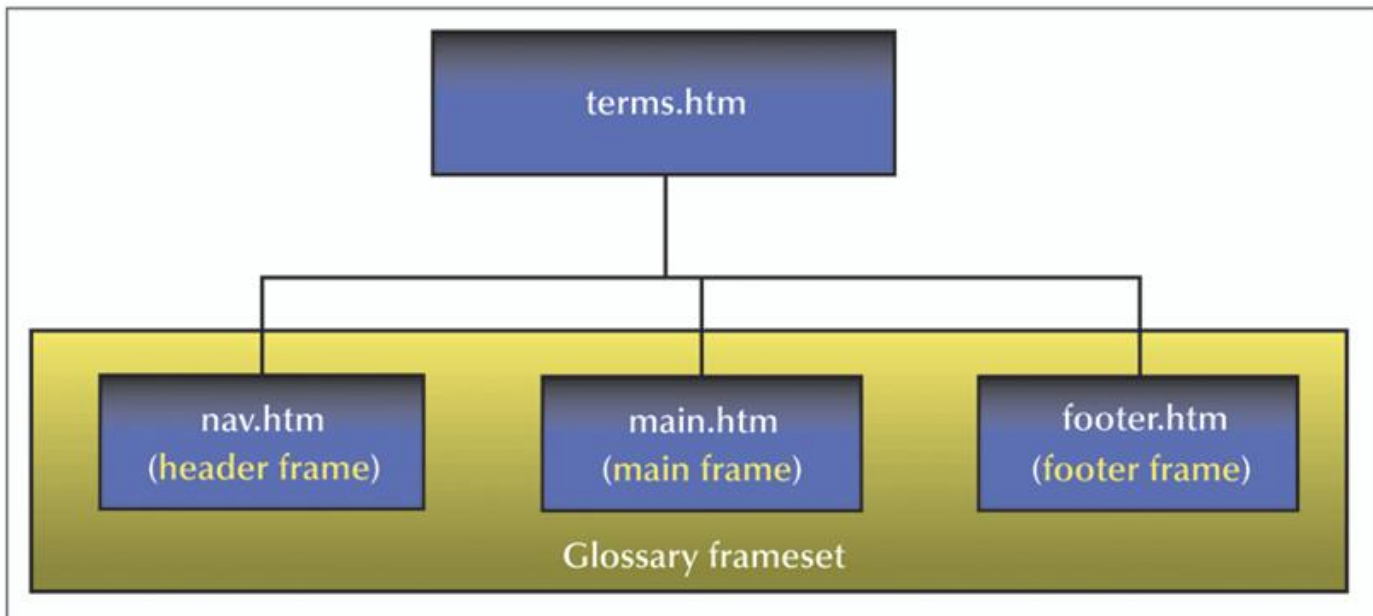
Working with Frames

- Working with the frame and frameset objects
 - Each frame in a frameset is part of the frames collection
 - `windowObject.frames[idref]`
 - To reference the header frame
 - `window.frames[0]`
 - `window.frames["header"]`



Working with Frames

- Navigating between frames
 - JavaScript treats the frames in a frameset as elements in a hierarchical tree





Working with Frames

- Navigating between frames
 - The **parent keyword** refers to any object that is placed immediately above another object in the hierarchy
 - If you want to go directly to the top of the hierarchy, you can use the **top keyword**



Working with Frames

- Treating frames as windows
 - In most cases JavaScript treats a frame as a separate browser window

```
frameObject.document.write(content)
frameObject.document.close()
frameObject.location.href = "url"
```




Working with Frames

- Setting the frameset layout
 - In JavaScript

```
frameset.rows = "text"  
frameset.cols = "text"
```



Working with Frames

- Collapsing and expanding a frame

```
<title>Glossary of Terms, iMusichistory</title>
<link href="main.css" rel="stylesheet" type="text/css" title="General" />

<script type="text/javascript">
function collapse() {
    parent.document.getElementById("Glossary").rows = "160,*,1";
}
function expand() {
    parent.document.getElementById("Glossary").rows = "160,*,64";
}
</script>
</head>
```



Working with Frames

- Collapsing and expanding a frame

```
<!-- Place Display/Hide Navigation Bars Here -->
<tr>
<td width="100" align="left">
<p class="gloss">
<a href="javascript:collapse()">Hide Navigation Bar</a></p>
</td>

<td bgcolor="white" class="body" valign="top" align="center">
<a class="gloss" name="glossary">Glossary of Music Terms</a></td>

<td width="100" align="right">
<p align="right" class="gloss">
<a href="javascript:expand()">Display Navigation Bar</a></p>
</td>
</tr>
<!-- End of Navigation Bars --->
```



Working with Inline Frames

- Another way to use frames in a Web site is by incorporating an inline frame

```
<iframe src="url" id="text" name="text"
width="value" height="value">alternate
content</iframe>
```

- You can reference it from the current document window using the object reference or as a frame using the frames reference



Tips for Working with Windows and Frames

- If you use JavaScript to write a transient status bar message, be sure to properly erase the message
- Keep the use of pop-up windows to minimum, and forewarn your users if possible
- Include code to verify that a pop-up window has not been blocked and, if possible, provide alternate methods



Tips for working with windows and frames

- Include code that makes it easy for users to close your pop-up windows
- Allow your users to resize your pop-up windows
- If the existence of pop-up blockers poses a problem, consider using alert, prompt, and confirm dialog boxes in place of pop-up windows



Tips for working with windows and frames

- If frames are a concern, add conditional statements to your documents to prevent them from appearing within the framesets of other Web sites
- Add JavaScript code to your frame documents so that they always appear within the context of their framesets

WINDOWS, FRAMES AND OVERLAYS

WINDOWS

- JavaScript's `Window` object represents the browser window, or potentially frame, that a document is displayed in.
- The properties of a particular instance of `Window` might include its size, amount of chrome—namely the buttons, scroll bars, and so on—in the browser frame, position, and so on.
- The methods of the `Window` include the creation and destruction of generic windows and the handling of special-case windows such as `alert`, `confirmation`, and `prompt` dialogs.

window Properties and Objects

Property	Description
ActiveXObject	Internet Explorer 9 returns a null value and says not to use it; however, previous Explorer versions do support this object.
applicationCache	Returns the application cache object for the window.
clientInformation	Object containing information about the user's browser and operating system.
closed	Boolean indicating if the Window object is closed.
constructor	Reference to the object's constructor.
content	Reference to the topmost Window object.
defaultStatus	The default message in the status bar.
dialogArguments	The argument(s) passed into <code>showModalDialog()</code> or <code>showModelessDialog()</code> .
dialogHeight	The height of a dialog window.
dialogLeft	The left position of a dialog window.
dialogTop	The top position of a dialog window.
dialogWidth	The width of a dialog window.
directories	This property is obsolete and is replaced by <code>personalbar</code> .
document	Reference to the Document object that allows manipulation of the elements in the page.

Property	Description
event	Object that holds information about the current event.
external	Reference to the external object that contains additional functionalities. The external object has many functions in Internet Explorer. Now the specification includes this object, though currently it only defines OpenSearch methods.
frameElement	The frame that the current window is embedded in.
frames[]	Array of frames in the page.
fullScreen	Boolean indicating if the browser is in full-screen mode.
globalStorage	Reference to a storage object that is used to store information across pages.
history	Reference to the History object that allows access to some data in the user's history, as well as methods for navigating the history.
Image	Creates a new Image object and returns a reference to the object.
innerHeight	The browser's client area height, including scroll bars.
innerWidth	The browser's client area width, including scroll bars.
length	The number of frames in the current page.
location	Reference to the Location object that returns information about the current URL and provides methods for manipulating the URL.

window Methods

Method	Description
<code>addEventListener()</code>	Registers an event handler for the <code>Window</code> object.
<code>alert()</code>	Displays an alert box.
<code>atob()</code>	Decodes base64 data.
<code>attachEvent()</code>	Registers an event handler on the <code>Window</code> object.
<code>back()</code>	Moves back one page in the user's history.
<code>blur()</code>	Removes the window from focus.
<code>btoa()</code>	Encodes a string in base64.

Method	Description
<code>clearInterval()</code>	Stops a currently running timer set up through <code>setInterval()</code> .
<code>clearTimeout()</code>	Stops a currently running timer set up through <code>setTimeout()</code> .
<code>close()</code>	Closes the window.
<code>confirm()</code>	Displays a confirmation dialog box.
<code>createPopup()</code>	Creates a pop-up window.
<code>detachEvent()</code>	Removes an event handler that was created using <code>attachEvent()</code> .
<code>dispatchEvent()</code>	Sends an event on the <code>Window</code> object.
<code>escape()</code>	Encodes a string by replacing some characters with their hex equivalent. Use <code>unescape()</code> to revert.
<code>execScript()</code>	Executes the script in the given language.
<code>find()</code>	Searches for the text in the document and highlights it if it is found.
<code>focus()</code>	Gives the window the focus.
<code>forward()</code>	Moves one page forward in the user's history.
<code>getComputedStyle()</code>	Gets the computed style for the given object.

Dialogs

- To create three types of special windows known generically as dialogs.
- A *dialog box*, or simply *dialog*, is a small window in a graphical user interface that “pop ups,” requesting some action from a user.
- The three types of basic dialogs supported directly by JavaScript include **alerts, confirms, and prompts**.

alert()

- The Window object's alert() method creates a special small window with a short string message and an OK button, as shown here:

-



The basic syntax for alert is

```
window.alert(string);
```

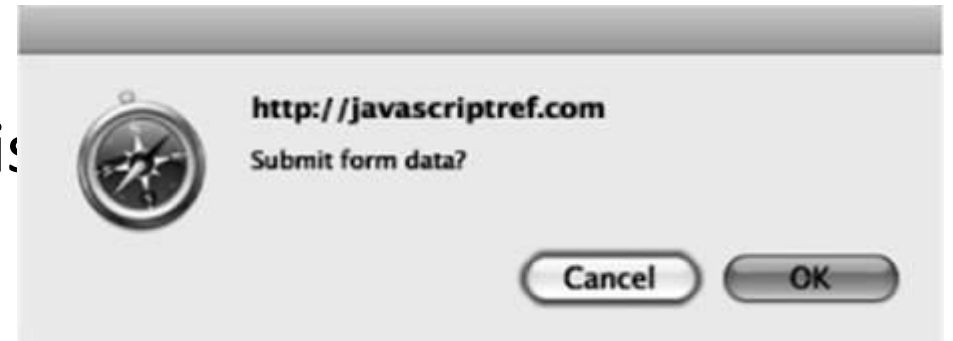
or, for shorthand, we just use

```
alert(string);
```

confirm()

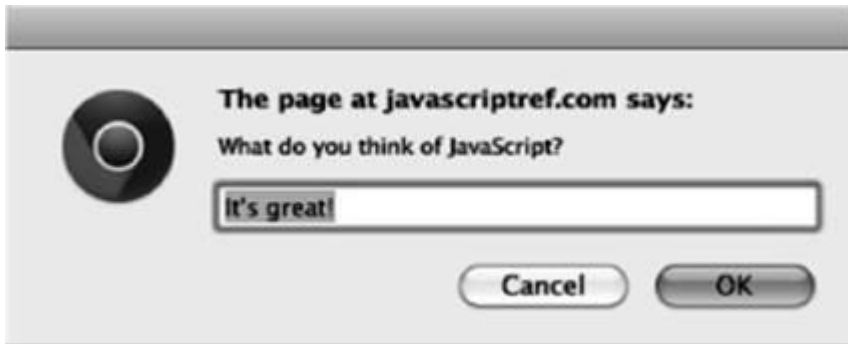
- The confirm() method creates a window that displays a message for a user to respond to by pressing either an OK button to agree with the message or a Cancel button to disagree with the message.
- A typical rendering is shown here:
- The basic syntax of the confirm() method is `confirm(string)` or simply,

```
confirm(string);
```



prompt()

- A prompt window invoked by the prompt() method of the Window object is a small data collection dialog that prompts the user to enter a short line of data, as shown here:



The prompt() method takes two arguments. The basic syntax is shown here:

```
resultvalue = window.prompt(prompt string, default value string);
```

- The first parameter is a string that displays the prompt value, and the second is a default value to put in the prompt window.
- The method returns a string value that contains the value entered by the user in the prompt.

The shorthand `prompt()` is almost always used instead of `window.prompt()`, and occasionally programmers will accidentally use only a single value in the method:

```
var result = prompt("What is your least favorite coding mistake?");
```

- However, in many browsers you may see that a value of `undefined` is placed in the prompt line.
- You should set the second parameter to an empty string to keep this from happening:

```
var result = prompt("What is your least favorite coding mistake?", "");
```

- When using the `prompt()` method, it is important to understand what is returned. If the user presses the Cancel button in the dialog or the close box, a **value of null will be returned**. It is always a good idea to check for this.
- Otherwise, a string value will be returned.
- Programmers should be careful to convert prompt values to the appropriate type using `parseInt()`, `parseFloat()`, or another type conversion scheme if they do not want a string value.

Opening and Closing Generic Windows

- While the `alert()`, `confirm()`, and `prompt()` methods create specialized windows quickly, it is often desirable to open arbitrary windows to show a Web page or the result of some calculation.
- The `Window` object methods `open()` and `close()` are used to create and destroy a `Window`, respectively.
- When you open a `Window`, you can set its URL, name, size, buttons, and other attributes, such as whether or not the window can be resized.
- The basic syntax of this method is

```
window.open(URL, name, features, replace)
```

- *URL* is a URL that indicates the document to load into the window.
- *name* is the name for the window (which is useful for referencing it later on using the **target** attribute of HTML links).
- *features* is a comma-delimited string that lists the features of the window.
- *replace* is an optional Boolean value (true or false) that indicates whether or not the URL specified should replace the window's contents. This would apply to a window that was already created.
- A simple example of this method is

```
var secondwindow = window.open("http://www.google.com", "google", "height=300,  
width=600, scrollbars=yes");
```

- This would open a window to Google with a height of 300 pixels, a width of 600 pixels, and scroll bars, as shown here:



- Once a window is open, the `close()` method can be used to close it.

Frames: A Special Case of Windows

- When a browser window contains multiple frames, it is possible to access each of the separate Window objects through `window.frames[]`, which is an array of the individual frames in the window.
- The basic properties useful for manipulating frames are,

Window Property	Description
<code>frames[]</code>	An array of all the <code>Frame</code> objects contained by the current window.
<code>length</code>	The number of frames in the window. Should be the same value as <code>window.frames.length</code> .
<code>name</code>	The current name of the <code>Window</code> . This is both readable and settable.
<code>parent</code>	A reference to the parent <code>Window</code> .
<code>self</code>	A reference to the current <code>Window</code> .
<code>top</code>	A reference to the top <code>Window</code> . Often the <code>top</code> and the <code>parent</code> will be one in the same unless a <code><frame></code> tag loads documents containing more frames.
<code>window</code>	Another reference to the current <code>Window</code> .

- The major challenge using frames and JavaScript is to keep the names and relationships between frames clear so that references between frames are formed correctly.

In this case, the window containing this document is considered the parent of the three frames (frame1, frame2, and frame5). While you might expect to use a value such as

`window.frames.length`

you probably will actually run the script from within a child frame to determine the number of frames in the window. Thus, you would actually use `window.parent.frames.length`

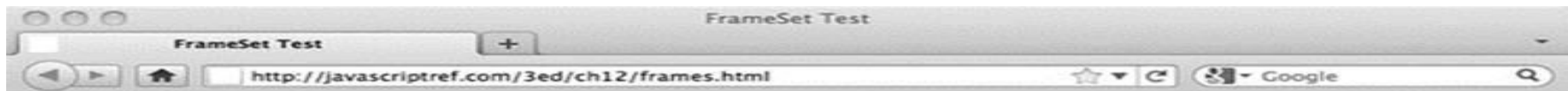
or just

```
parent.frames.length
```

- The parent property allows a window to determine the parent window.
- To access a particular frame, we can use both its name as well as its position in the array, so the following would print out the name of the first frame, which in our case is “frame1”:

```
parent.frames[0].name
```

- We could also access the frame from another child frame using `parent.frame1`, or even `parent.frames[“frame1”]`, using the associate array aspect of an object collection.
- Remember that a frame contains a window, so once you have this you can then use all of the Window and Document methods on what the frame contains.



Window: frame1

Current window has no frames directly within it

```
parent.frames.length = 3  
parent.frames[0].name = frame1  
parent.frames[1].name = frame2  
parent.frames[2].name = frame5
```

```
top.frames.length = 3  
top.frames[0].name = frame1  
top.frames[1].name = frame2  
top.frames[2].name = frame5
```

Window: frame3

Current window has no frames directly within it

```
parent.frames.length = 2  
parent.frames[0].name = frame3  
parent.frames[1].name = frame4
```

```
top.frames.length = 3  
top.frames[0].name = frame1  
top.frames[1].name = frame2  
top.frames[2].name = frame5
```

Window: frame4

Current window has no frames directly within it

```
parent.frames.length = 2  
parent.frames[0].name = frame3  
parent.frames[1].name = frame4
```

```
top.frames.length = 3  
top.frames[0].name = frame1  
top.frames[1].name = frame2  
top.frames[2].name = frame5
```

Window: frame5

Current window has no frames directly within it

```
parent.frames.length = 3  
parent.frames[0].name = frame1  
parent.frames[1].name = frame2  
parent.frames[2].name = frame5
```

```
top.frames.length = 3  
top.frames[0].name = frame1  
top.frames[1].name = frame2  
top.frames[2].name = frame5
```


Inline Frames

- One variation of frames that deserves special attention is the **<iframe>**, or inline frame, because it is preserved under HTML5. The idea with an inline frame is that you can add a frame directly into a document without using a frameset. For example, this example

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Iframe</title>
</head>
<body>
<h1>Regular Content here</h1>
<iframe src="http://www.google.com" name="iframe1" id="iframe1" height=
"200" width="400"></iframe>
<h1>More content here</h1>
</body>
</html>
```

produces a page something like this:



Regular Content here



More content here

- **Applied Frames**

- Now that we are familiar with frame naming conventions, it is time to do something with them. In this section, we present some solutions for common frame problems and hint at the larger issues with frame usage.

- **Loading Frames**

- A common question developers have with HTML is how to load multiple frames with a link. XHTML provides the target attribute to target a single frame, such as *framename*, like so:

```
<a href="http://www.google.com" target="framename">Google</a>
```

Frame Busting

- While frames can be very useful for building some complex user interfaces and comparing documents, they also can cause Web designers significant problems.

Frame Building

- The converse problem to the one solved by frame busting would be to avoid having framed windows displayed outside of their framing context. This occasionally happens when users bookmark a piece of a frameset or launch a link from a frameset into a new window. The basic idea would be to have all framed documents make sure they are inside of frames by looking at each window's location object, and if they are not, to dynamically rebuild the frameset document.

Overlays Instead of Windows

- Simple dialogs such as `alert()` and `prompt()` lack customization.
- You may opt to try to create custom dialogs using the generic `window.open()` method.
- However, in either case, the dialogs may be blocked by browser-based or third-party pop-up blockers installed by the user.
- To address both the customization concerns and pop-up blockers, many designers have turned to what we dub “div dialogs,” named for the HTML `<div>` tag used to create them.
- Using CSS, designers can position `<div>` tag–based regions over content and customize them visually in whatever manner they like.