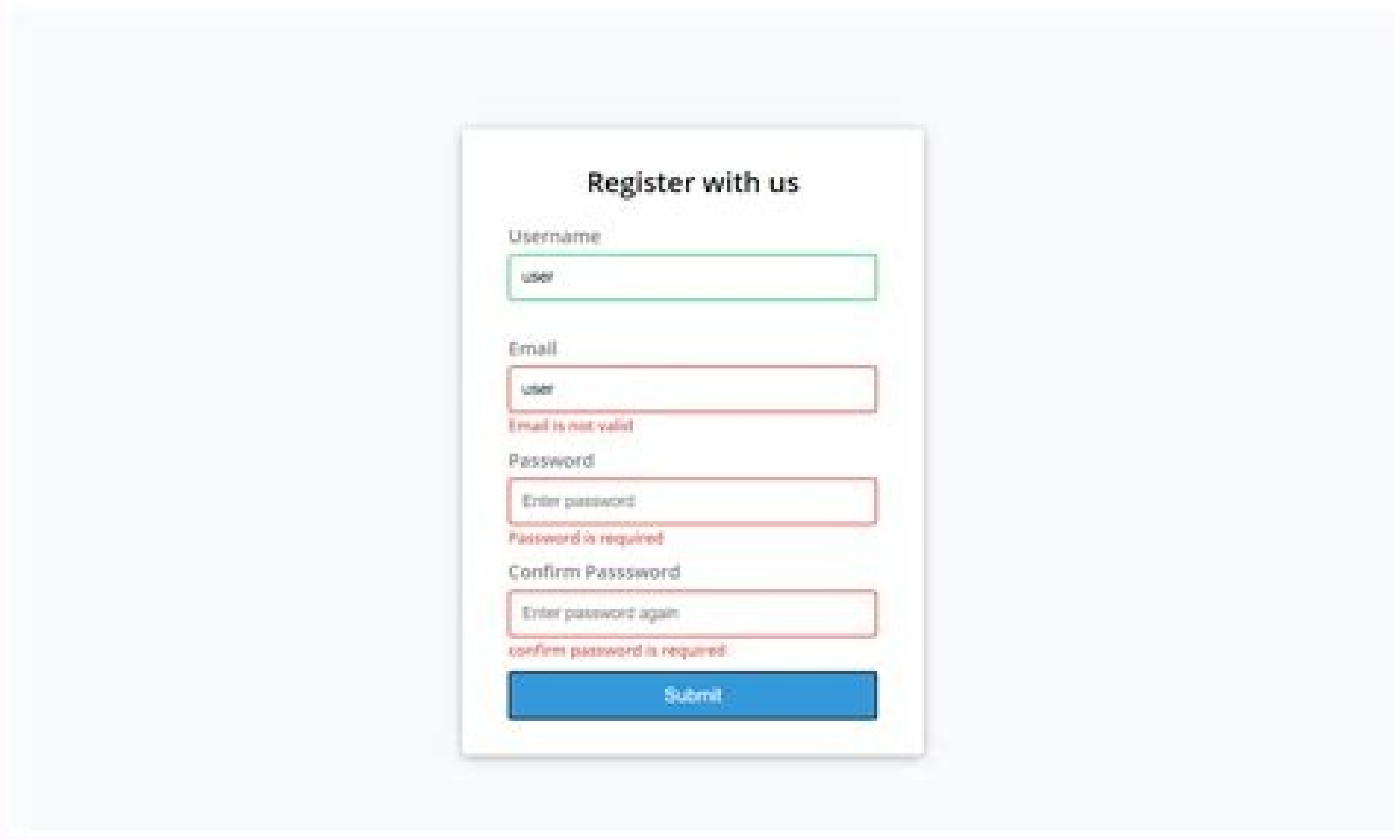
Javascript prevent form submit until validation

I'm not robot!


Register with us

Username

user

Email

user

Email is not valid

Password

Enter password

Password is required

Confirm Password

Enter password again

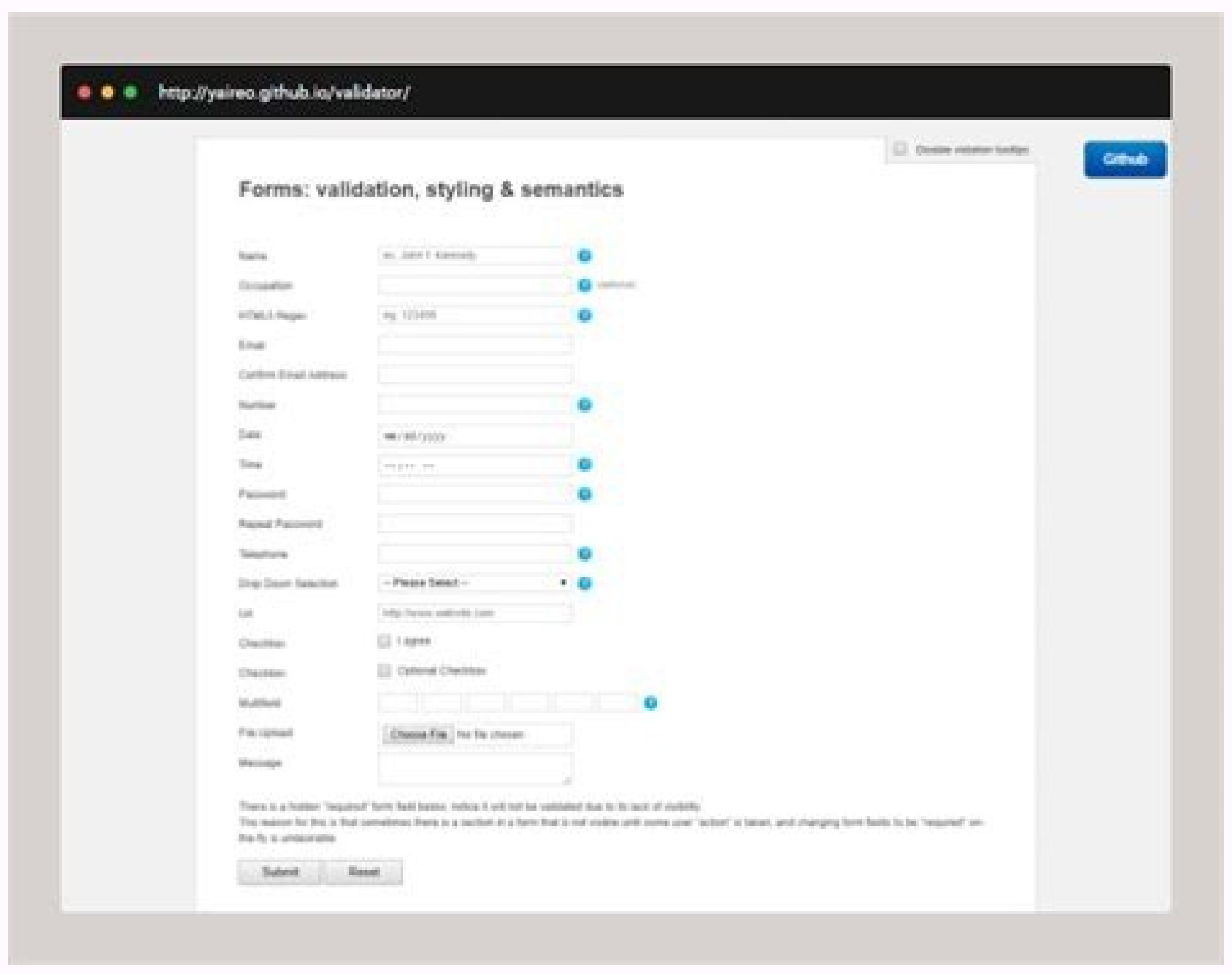
confirm password is required

Submit

Registration Form

User id:	Required and must be of length 5 to 12.	
Password:	Required and must be of length 7 to 12.	
Name:	Required and alphabates only.	
Address:	Optional.	
Country:	(Please select a country)	Required. Must select a country.
ZIP Code:	Required. Must be numeric only.	
Email:	Required. Must be a valid email.	
Sex:	○ Male ○ Female	Required.
Language:	✓ English ☐ Non English	Required.
About:	Optional.	

Submit


http://yaireo.github.io/validator/

Github

Forms: validation, styling & semantics

Name

Occupation

HTML5 Regex

Email

Confirm Email Address

Number

Date

Time

Password

Repeat Password

Telephone

Drop Down Selection

-- Please Select --

Url

Checkbox

I agree

Checkbox

Optional Checkbox

Multifield

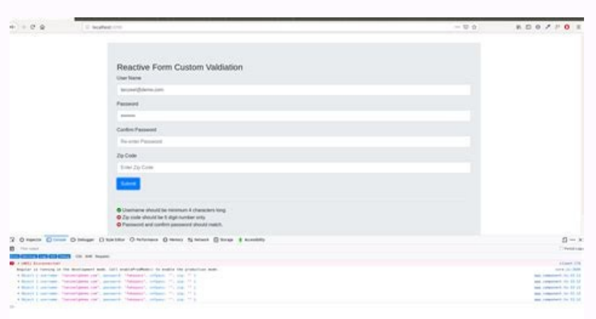
File Upload

Choose File

Message

[illegible]

Submit Reset


Reactive Form Custom Validation

Name	input is not complete
Occupation	(optional) please put something here
HTML5 Regex	please put something here
Email	please put something here
Confirm Email Address	please put something here
Number	please put something here
Date	please put something here
Password	please put something here
Repeat Password	please put something here

Javascript form validation rules. Javascript stop form submit if validation fails. Javascript form validation drop down list.

I am making form and I have some problems validating it. HTML: Job Pocket Javascript: $("#regForm").submit(function(event){ if(document.getElementById("email")=="" || document.getElementById("password") || document.getElementById("last_name") || document.getElementById("first_name")) return false; }); Even with this javascript code added the form still submits to signup.php. And should I add extre checks in php aswell. 2 Sometimes, we want to prevent form submission in a React component. In this article, we'll look at how to prevent form submission in a React component. To prevent form submission in a React component, we should call the event.preventDefault method in our submit event handler function. For instance, we write: import React from "react"; export default function App() { const onSubmit = (event) => { event.preventDefault(); console.log("submission prevented"); }; return ( ); } We have a form with the onSubmit prop set to the onSubmit function so that the onSubmit function is our form's submit event handler. In the function, we call event.preventDefault to prevent the default submission behavior. Therefore, when we click Submit, we see the 'submission prevented' message logged in the console. Conclusion To prevent form submission in a React component, we should call the event.preventDefault method in our submit event handler function. Related Posts This post will discuss how to prevent an HTML form from being submitted in JavaScript and jQuery. The simplest solution to prevent the form submission is to return false on submit event handler defined using the onsubmit property in the HTML element. Choose your preferred contact method: Email Phone Submit Edit in JSFiddle Alternatively, you can call the Event.preventDefault() to prevent the default action on a form submission from executing. Choose your preferred contact method: Email Phone Submit Edit in JSFiddle Note that it is always a good practice to separate JavaScript from HTML markup, but the above solution fails to do so. A better solution is to bind an event handler to the "submit" JavaScript event. The submit event fires whenever the user submits a form. $(document).ready(function() { $('form').submit(function(e) { Visually the form looks like this: I have done some animation through the focus on the input and the validation, wrote it and decided to finished button, in the end, I have sent the request without passing the validation with the help Js'and. JavaScript code:Html form code Orange button which when pressed is sent. Tried to implement using event.preventDefault() and using return validate() / return false. HTML form elements are the foundation for web page interactions and they improved quite a little bit over the last years. Today, developers can use different types (number, tel, color, ...) and set different input modes (text, decimal, email, ...) to name only two examples.What remained complicated was to submit forms via JavaScript. Even though the HTMLFormElement defines a submit method, it does not quite behave as one would expect.Let's assume we have the following HTML form: Your name Submit And some JavaScript:document.querySelector('form') .addEventListener('submit', (event) => { event.preventDefault(); console.log('submitted form'); }); When one clicks the submit button the following happens:the form is validated and possible errors are shownif the form validation passes and the form is valid, it fires a submit eventthe submit handler is called and it prevents the form submission due to event.preventDefault()The triggered submit event gives developers a way to react to form submissions in JavaScript. And it's used a lot! A common scenario is to prevent form submissions (preventDefault), and make AJAX requests to replace page contents dynamically.But what happens when you submit a form in JavaScript via submit?document.querySelector('form').submit(); The answer is – the form is submitted! (🤷‍♂ duh!) What's surprising is that there won't be input and form validation, and there won't be a submit event. All the field values are submitted no matter if the inputs are valid or not.This is unexpected behavior and it should behave like pressing the submit button. There are surely reason for skipping the validation, but I'd expect that submit also validates the form and only proceeds if everything is valid.You can work around this problem by triggering the click on the submit button. The click then triggers the standard behavior that users see when they interact with a form, including validations and a fired submit event.Mimicking user behavior works fine and that's great, but I never thought of this solution as elegant. It turns out there's a better way!People started to work on a solution to this behavior in June 2019 (the proposal is an interesting read). The HTMLFormElement now includes a new method called requestSubmit. And this method does the same as clicking a submit button. There is no magic to it – the JavaScript method does what you expect and offers the great goodies HTML forms ship by default (including the form validation). I have to say – I'm excited about it!submitrequestSubmitdoesn't trigger submit eventtriggers submit eventdoesn't trigger form validationtriggers form validationcan't be canceledcan be canceled via event.preventDefault in a submit event handlerThe method's browser support as of March 2021 is as follows:Update: Safari 16 starts shipping requestSubmit so that cross-browser support will be given soon.You can read more about the requestSubmit method on MDN, dive into its specification or see it in action on CodePen.You can see a #devsheet visualizing the difference in the video below.requestSubmit show of (devsheet)If you're interested in reading more quick TIL ("today i learned") posts, subscribe to my weekly newsletter. NN 2, IE 3 8.3.1 Problem You want a validation function that detects incorrect data entry to halt the submission of the form until the user corrects the data entry. 8.3.2 Solution Batch validation checking typically operates from the onsubmit event handler of the form element. Submission is aborted if the event handler evaluates to return false. Include the return statement in the event handler assignment, and let the validation function supply the Boolean value based on its findings: 8.3.3 Discussion You can implement batch validation by way of a master function that calls the individual validation functions as needed. To demonstrate, let's create a small form with numerous control types in it. Text fields execute real-time validation, while the form's onsubmit event handler performs the batch validation. Both validation types use the validation functions shown in Recipe 8.2 (either the string parsing or regular expression varieties). Here's the form's HTML: First Name:
Last Name:
Email Address:
Your Region: Choose One: Africa Asia Australia/Pacific Europe North America South America
Licensing Terms: I agree I do not agree
You can see the form in Figure 8-1. As the user tabs and clicks through the form, typically the only validation taking place is in the email text box. Tabbing through the empty name fields without making any changes won't trigger the onchange event handlers there (another reason why batch validation is needed). In this form, we also want to make sure that a choice is made from a select element, and that a member of the radio button group is clicked (some designers might question delivering radio buttons without a default selection, but this example requires no initial selection). When the user clicks the Submit button, the validateForm( ) function executes to perform validations of all required form controls. The calls to the validations are cascaded so that if there are multiple errors and the user corrects the first one to be reported, subsequent clicks of the Submit button find an error lower in the form than a previous one: function validateForm(form) { if (isNotEmpty(form.name1)) { if (isNotEmpty(form.name2)) { if (isNotEmpty(form.eMail)) { if (isEMailAddr(form.eMail)) { if (isChosen(form.continent)) { if (isValidRadio(form.accept)) { return true; } } } } } } return false; } Validation functions for a select element and radio button group are not among routines in Recipe 8.2, but are shown here: // validate that the user made a selection other than default function isChosen(select) { if (select.selectedIndex = = 0) { alert("Please make a choice from the list."); return false; } else { return true; } } // validate that the user has checked one of the radio buttons function isValidRadio(radio) { var valid = false; for (var i = 0; i < radio.length; i++) { if (radio[i].checked) { return true; } } alert("Make a choice from the radio buttons."); return false; } Note that the select element design assumes that the first item is an invalid choice. All of the functions feed back to the main dispatching function. If any one validation fails, the dispatching function returns false to the onsubmit event handler, forcing it to evaluate to return false and thus aborting the submission. But if all validations return true, the dispatching function also returns true to the event handler, allowing the submission to continue normally. Relying on the onsubmit event handler means that the user could disable JavaScript in the client to bypass client-side validation. This is a good reason to duplicate validation on the server. But if you'd rather perform all validation on the client and you know that all users have scriptable browsers, consider using a button-type input element instead of a true submit-type input element. Let the button's onclick event handler invoke the batch validation function and (if validation succeeds) the submit( ) method of the form object. Under these conditions, the user can't submit the form with JavaScript turned off. 8.3.4 See Also Recipe 8.2 for individual data validation functions; Recipe 8.4 for focusing and selecting text in an invalid text field; Recipe 2.12 for date field validation suggestions. Page 2 NN 2, IE 3 8.2.1 Problem You want to verify that a text box contains one of the following: any text, a number, a string of a fixed length, or an email address. 8.2.2 Solution Apply the library of text field validation routines shown in the Discussion (Example 8-1 and Example 8-2) to your form. The library includes the following functions: isNotEmpty( ) The field has one or more characters in it. isNumber( ) The value is a number. isLen16( ) The field contains exactly 16 characters. isEMailAddr( ) The field contains a likely email address. For real-time validation of text box entries, use an onchange event handler in the input element and pass a reference to the element by way of the this keyword. For example, the following input element could be used for an email address: See Recipe 8.3 for an example of how these validation functions can be linked together in batch validation prior to submitting the form. The return values from the validation functions are vital for successful operation triggered by the form's onsubmit event handler. 8.2.3 Discussion Example 8-1 shows a set of fully backward-compatible text validation functions. All of these functions are to be invoked by both the onchange event handler of the text box and a batch validation function triggered by the onsubmit event handler of the enclosing form. All functions are passed references to the form control invoking the event handler. Example 8-1. Backward-compatible text field validation functions // validates that the field value string has one or more characters in it function isNotEmpty(elem) { var str = elem.value; if(str = = null || str.length = = 0) { alert("Please fill in the required field."); return false; } else { return true; } } // validates that the entry is a positive or negative number function isNumber(elem) { var str = elem.value; var oneDecimal = false; var oneChar = 0; // make sure value hasn't cast to a number data type str = str.toString( ); for (var i = 0; i < str.length; i++) { oneChar = str.charAt(i).charCodeAt(0); // OK for minus sign as first character if (oneChar = = 45) { if (i = = 0) { continue; } else { alert("Only the first character may be a minus sign."); return false; } } // OK for one decimal point if (oneChar = = 46) { if (!oneDecimal) { oneDecimal = true; continue; } else { alert("Only one decimal is allowed in a number."); return false; } } // characters outside of 0 through 9 not OK if (oneChar < 48 || oneChar > 57) { alert("Enter only numbers into the field."); return false; } } return true; } // validates that the entry is 16 characters long function isLen16((elem) { var str = elem.value; if (str.length != 16) { alert("Entry does not contain the required 16 characters."); return false; } else { return true; } } // validates that the entry is formatted as an email address function isEMailAddr(elem) { var str = elem.value; str = str.toLowerCase( ); if (str.indexOf("@") > 1) { var addr = str.substring(0, str.indexOf("@")); var domain = str.substring(str.indexOf("@") + 1, str.length); // at least one top level domain required if (domain.indexOf(".") = = -1) { alert("Verify the domain portion of the email address."); return false; } // parse address portion first, character by character for (var i = 0; i < addr.length; i++) { oneChar = addr.charAt(i).charCodeAt(0); // dot or hyphen not allowed in first position; dot in last if ((i = = 0 && (oneChar = = 45 || oneChar = = 46)) || (i = = addr.length - 1 && oneChar = = 46)) { alert("Verify the user name portion of the email address."); return false; } // acceptable characters (- . _ 0-9 a-z) if (oneChar = = 45 || oneChar = = 46 || oneChar = = 95 || (oneChar > 47 && oneChar < 58) || (oneChar > 96 && oneChar < 123)) { continue; } else { alert("Verify the user name portion of the email address."); return false; } } for (i = 0; i < domain.length; i++) { oneChar = domain.charAt(i).charCodeAt(0); if ((i = = 0 && (oneChar = = 45 || oneChar = = 46)) || ((i = = domain.length - 1 || i = = domain.length - 2) && oneChar = = 46)) { alert("Verify the domain portion of the email address."); return false; } if (oneChar = = 45 || oneChar = = 46 || oneChar = = 95 || (oneChar > 47 && oneChar < 58) || (oneChar > 96 && oneChar < 123)) { continue; } else { alert("Verify the domain portion of the email address."); return false; } } return true; } alert("The email address may not be formatted correctly. Please verify."); return false; } Regular expression versions of the validation functions are more compact when the validation is complex (as in the case of email addresses), but they require great care and stress testing to make sure they are doing what you expect. Example 8-2 shows the equivalent validation functions using regular expressions. Example 8-2. Text field validation functions with regular expressions // validates that the field value string has one or more characters in it function isNotEmpty(elem) { var str = elem.value; var re = /.+/; if(!str.match(re)) { alert("Please fill in the required field."); return false; } else { return true; } } //validates that the entry is a positive or negative number function isNumber(elem) { var str = elem.value; var re = /^[-]?\d*\.?\d*$/; str = str.toString( ); if (!str.match(re)) { alert("Enter only numbers into the field."); return false; } return true; } // validates that the entry is 16 characters long when // input field's maxlength attribute is set to 16 function isLen16(elem) { var str = elem.value; var re = /\b.{16}\b/; if (!str.match(re)) { alert("Entry does not contain the required 16 characters."); return false; } else { return true; } } // validates that the entry is formatted as an email address function isEMailAddr(elem) { var str = elem.value; var re = /^[\w-]+(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]{2,7}$/; if (!str.match(re)) { alert("Verify the email address format."); return false; } else { return true; } } Notice that the validation done in these functions provides the user with less detailed information about the more complex data entries than the backward-compatible versions. It is possible to provide more information, but this involves pulling apart the regular expressions to test for subsets of matches. In the first one, isNotEmpty( ), the regular expression pattern looks for a string with one or more characters of any kind (.+). To test for a number in isNumber( ), the pattern looks for a string that begins with (^) zero or one minus signs ([-]?), followed by zero or more numerals (\d*), zero or one decimals (\.?), and zero or more numerals (\d*) on the tail end ($). A fixed-length string pattern in isLen16( ) looks for word boundaries on both ends (\b) and any characters (.) appearing 16 times ({16}). To ensure the user keeps to the 16-character length, limit the text-type input element to a maximum length of 16. The gnarled email pattern inside isEMailAddr( ) looks for a match that begins (^) with one or more letters, numerals, underscores, or hyphens ([\w-]), followed by zero or more combinations of a period, letter, numeral, underscore, or hyphen ((\.[\w-]+)*), followed by the @ sign, followed by one or more computer or domains (([\w-]+\.)+), followed by two to seven upper- or lowercase letters for the top-level domain name ([a-zA-Z]{2,7}) on the tail end ($). The pattern does not match the use of straight IP addresses for the portion after the @ sign, but the email message specification (Internet Engineering Task Force RFC 822) frowns on such usage anyway. In addition to individual validation routines, you sometimes need to cascade them. For example, none of the functions that validate numbers, fixed-length strings, or email addresses perform any checking that assures the field has something in it. For example, if the email address field is a required field in the form, you would wire the onchange event handler for that input element to pass the values first to the isNotEmpty( ) function and then the isEMailAddr( ) function�but in such a way that if the first fails, the second one does not execute. That's where the returned Boolean values of the functions come into play: Not shown among the text field validation routines here is one that validates a date entry. Validating date entries is tricky business due to the wide range of date formats and sequences of numbers used around the world. Except for intranet application where everyone standardizes on a single date format, I recommend implementing date input as three distinct fields (or select elements) for entry of month, date, and year. Use the onsubmit event handler to combine the values into a single string for a hidden input element whose value the server can pass directly to the database. You can also use the pop-up calendar shown in Recipe 15.9 to help a user select a date, leaving the formatting up to your scripts. Or, if all of your users follow a fixed date format, you can try the date validation techniques described in Recipe 2.12. 8.2.4 See Also Recipe 8.3 for a batch validation structure that uses the functions just described; Recipe 8.4 for automatically focusing and selecting a text field that fails validation; Recipe 2.12 for date validation ideas. Page 3 NN 2, IE 3 8.1.1 Problem You want the user to begin typing into the first text box of a form without having to physically bring focus to the box. 8.1.2 Solution Assuming that the page always contains a form and text field with the same names, use an onload event handler to invoke the focus( ) method of the text box: Substitute the name of your form and text box for the two placeholders in this code. 8.1.3 Discussion As simple as this solution is, you'll be amazed how much it helps casual users get faster results from a page's form. The Google search form uses a similar technique so that all a user needs to do is select the google.com site from the Favorites/Bookmarks list and start typing the query string. Without this scripted assistance, users of most browsers have to click on the text box or press the Tab key a couple of times to bring focus to the text box. You must wait for the page to complete its loading process before giving the text box focus, or some browsers will grab focus away from the box. If you deliver the text box with default or sample text that is to be replaced by the user, you should also invoke the select( ) method of the box: This action also selects the text in the box, such that the next keyboard key that gets pressed removes the default text. 8.1.4 See Also Recipe 8.4 for auto-selecting a form field that fails validation; Recipe 8.10 for auto-tabbing between text boxes with fixed-length entries. Page 4 Giving scripted intelligence to web forms was the impetus that led to the development of the JavaScript language and the notion of a document object model. While a lot has happened to scripting in the meantime, forms still make frequent use of scripts to assist with user-friendly instantaneous interaction that otherwise requires a two-way trip to the server (and delays for the user) to accomplish. Because of the comparatively long history of scriptable forms and form controls, it is comforting to know that most such scripts work with a wide range of browsers, and not just those that implement the W3C DOM. Even so, there are some misunderstandings about the combination of scripts and forms that I'll attempt to clear up in this chapter. 8.0.1 Referencing Forms and Controls Before the W3C DOM, scripts used what is now known as DOM Level 0 syntax to reference form objects and the form controls (input and textarea elements) within them. This long-time convention relies for the most part on the form and controls having name attributes assigned to them. In fact, even today's browsers won't submit form control values to the server unless the

elements have names assigned to them (independent of the now ubiquitous id attribute). At the same time, however, the object model provides arrays of forms and form elements, which can be accessed through JavaScript array syntax and numerical index values. For example, if a document contains a single form whose name is userInfo, backward-compatible scripts can reference the form object in any of the following ways: document.forms[0] document.forms["userInfo"] document.userInfo Each form element object also contains an elements array that contains references to all of the recognized form controls nested inside the frame. For example, if the second input element of the userInfo form is a text box named age, you have three ways to reference that text box for each of the three ways you can use to reference the containing form. Using just one containing form reference, here is an example of three equivalent references to the age text box: document.userInfo.elements[1] document.userInfo.elements["age"] document.userInfo.age Notice how this syntax follows the element containment hierarchy: document to form to control. This allows for the possibility of a form control's name being reused in multiple forms on the page�something not possible (or at least not encouraged) with id attributes. In browsers supporting scriptable id attributes of elements (IE 4 or later and NN 6 or later), you can also reference a form directly by way of the object model syntax(es) supported by the browser. For example, in IE 4 and later, you can use the Microsoft DOM reference syntax: document.all.formID document.all.formControlID For W3C DOM syntax (IE 5 or later and NN 6 or later), use the regular element-referencing syntax: document.getElementById("formID") document.getElementById("formControlID") Even though your scripts can use only the ID to build references, you'll want to assign an identifier to both the name and id attributes of each element if the form is to be submitted to the server. You can use the same identifier for both attributes of an element and not risk collisions. Browser versions that you need to support with your scripts should dictate the syntax you use to address forms and controls. If backward-compatibility is of any concern with your audience (including Navigator 4), stick with the DOM Level 0 syntax. It will continue to be supported in new mainstream browsers for a long time to come. 8.0.2 Form Validation Strategies Client-side form validation is a helpful service that speeds the correction of potential errors in forms before they ever reach the server. That is not meant to imply that client-side validation can replace server-side validation. Far from it. But, like most DHTML applications, client-side validation helps your users be more efficient when filling out complex forms. Even on the client, however, you have two types of validation strategies to consider: real-time and batch mode. In real-time validation, a script looks for signs of activity�such as onchange events in text boxes�to immediately validate an entry against whatever data restrictions apply to the field. The advantage of instantaneous validation feedback is that the user's mind is still fresh about the information filled into a field. In other words, when the user has just entered an address into an email address field, it's more helpful to bring the user immediately back to that field and correct the error, rather than wait until later. On the other hand, in most situations, you should not be so dogmatic as to absolutely require that a form be filled out in order, which can happen when you use onblur events to trigger validation. Give the user a chance to tab through and skip over a text box while filling out a form. But then be sure to catch any missed or passed text boxes with a batch validation right before the form gets submitted. This offers a last-chance review of data before sending it to the server. In most cases, the same routines you use for the real-time field checking can be reused by the batch validation routines triggered either by the onsubmit event of the form object or by a regular button-type input element that invokes validation routines and form submission. Additionally, you will have to decide whether your text box inspections will use old-fashioned string parsing or the more modern regular expression facilities of JavaScript. Regular expressions provide powerful and quick ways of looking for patterns in a text box entry, but the syntax for using regular expressions is rather terse and cryptic. You will see some examples of both forms in Recipe 8.2, and the introduction to Chapter 1 presents a brief overview of regular expressions in JavaScript, but you'll need a solid JavaScript tutorial or reference to understand the full scope of regular expressions. 8.0.3 Email Submissions and Return Pages A traditional HTML form submits its data to a program running on the server. That program can be in a variety of languages (Perl, Java, Visual Basic, C, C++, C#, Tcl, and more), but the client never knows or cares what happens on the server. In typical operation, the server program receives the data, tears it apart, stuffs it into database tables, or otherwise manipulates the data for storage on the server. When the data is processed correctly, the server then returns an HTML page back to the client. By default, that return page displays in the same window or frame as the form. There's nothing particularly special about this transaction. But many scripters don't have access to server scripting, either because they host on closed systems, or the technology is beyond their field of expertise. To fill in the gap and still capture the data submitted by a form, they have resorted to assigning a mailto: URL to the action attribute of a form element. When this approach first appeared in browsers, the form's data was quietly emailed to the address assigned to the mailto: URL. But as security consciousness overtook the Web (especially with the advent of scripting), such surreptitious emailing came to an end. Instead, a more explicit email process took hold, whereby the user with a properly configured browser and email client sees the email message before it is sent. A problem with this scenario is that a significant number of users either don't have their email clients configured correctly and matched to their browser, or they are intimidated by the appearance of the email window. The result is that using a mailto: URL to submit forms is less likely to capture all submissions that you might otherwise receive. However, should you persist with your usage of the mailto: URL, be aware that this kind of submission does not return any kind of page to the browser. There is no confirmation that the mail was submitted correctly or completely. Moreover, there is no absolutely reliable way to script a dummy confirmation page. To do so reliably requires that the browser receive some notification of completion so that the page could be displayed in place of the form's page. If you or a script removes the form's page before the actual submission completes its task, the submission may be aborted on the network�losing even more potential submissions before they reach you. The solution is to search the Web for a third-party host for a Unix program called FormMail. This program lets you submit a form to a genuine server program that forwards the content to a mail address you supply in the setup process. Page 5 Section 8.0. Introduction Section 8.1. Auto-Focusing the First Text Field Section 8.2. Performing Common Text Field Validations Section 8.3. Preventing Form Submission upon Validation Failure Section 8.4. Auto-Focusing an Invalid Text Field Entry Section 8.5. Changing a Form's Action Section 8.6. Blocking Submissions from the Enter Key Section 8.7. Advancing Text Field Focus with the Enter Key Section 8.8. Submitting a Form by an Enter Key Press in Any Text Box Section 8.9. Disabling Form Controls Section 8.10. Hiding and Showing Form Controls Section 8.11. Allowing Only Numbers (or Letters) in a Text Box Section 8.12. Auto-Tabbing for Fixed-Length Text Boxes Section 8.13. Changing select Element Content Section 8.14. Copying Form Data Between Pages Page 6 NN n/a, IE 5.5(Win) 7.9.1 Problem You want to use a script to build an entirely new set of frames within the current frameset. 7.9.2 Solution Use either of the makeNewFrameset( ) functions described in the Discussion to empty the current set of frames and install a new set. The current framesetting document's scripts and variables remain intact throughout the process. Invoke the function from a user interface element (button or link) or from any other function in one of the frames: parent.makeNewFrameset( ); 7.9.3 Discussion The makeNewFrameset( ) function shown in Example 7-2 converts a frameset that is initially delivered as one set up in two columns to a frameset with different documents in a two-row frameset. Due to idiosyncrasies in a variety of browsers that should support this functionality, the function operates only in IE 5.5 or later for Windows. Example 7-2. A function to generate a new frameset for IE 5.5 or later function makeNewFrameset( ) { var newFrame1 = document.createElement("frame"); newFrame1.id = "newFrame1"; newFrame1.name = "newFrame1"; newFrame1.src="altNavBar.html" var newFrame2 = document.createElement("frame"); newFrame2.id = "newFrame2"; newFrame2.name = "newFrame2"; newFrame2.src="altHome.html" var frameset = document.getElementById("masterFrameset"); while (frameset.childNodes.length > 0) { frameset.removeChild(frameset.firstChild); } frameset.cols = null; frameset.rows = "80, *"; frameset.appendChild(newFrame1); frameset.appendChild(newFrame2); } It is important to clean out the existing node tree and associated properties before repopulating it with new elements. In this function, the new frame elements are created in memory only. While they stand by, the original frameset is first cleansed of its previous child nodes. Thus, even if the main frameset has a more complex frame construction inside it, everything is gone after the while loop does its job. Because we're changing the main frameset from a column- to row-oriented frameset, the original setting for the cols attribute is cleared out before assigning fresh values to the rows property. At last, the new frame elements are appended to the frameset. If IE 6 is your only target browser for this kind of application, you might consider using a W3C DOM DocumentFragment object as a temporary container for the frame elements. The revised function is shown in Example 7-3. Example 7-3. A function to generate a new frameset for IE 6 or later function makeNewFrameset( ) { var frag = document.createDocumentFragment( ); var newFrame= document.createElement("frame"); newFrame.id = "newFrame1"; newFrame.name = "newFrame1"; newFrame.src="altNavBar.html" frag.appendChild(newFrame); newFrame = document.createElement("frame"); newFrame.id = "newFrame2"; newFrame.name = "newFrame2"; newFrame.src="altHome.html" frag.appendChild(newFrame); var frameset = document.getElementById("masterFrameset"); while (frameset.childNodes.length > 0) { frameset.removeChild(frameset.firstChild); } frameset.cols = null; frameset.rows = "30%, *"; frameset.appendChild(frag); } For a backward-compatible solution, you can always fall back on the document.write( ) method of the parent or top frame. Assemble the entire content of a new framesetting document as one string. Then issue one top.document.write( ) method followed by a top.document.close( ) method. All vestiges of the original frameset (including script variables in the frameset and frames) disappear, and are replaced by the new frameset. 7.9.4 See Also Recipe 7.8 for resizing frames; Recipe 14.2 for using document.write( ) to generate a new page (and techniques for including script libraries in such a page). Page 7 NN 6, IE 4 7.8.1 Problem You want a script to adjust the size of one or more frames in a frameset, including resizing a frame to zero width or height to hide a frame. 7.8.2 Solution Apply a variation of the script shown in Example 7-1 in the Discussion to your frameset (customized with your frameset element's ID). The script goes in the framesetting document and is invoked by a user interface element in one of the frames. For example, a button specification in one of the frames invokes the toggleFrame( ) function as follows: Hide/Show Navbar You could also use hyperlinks or linked images to act as clickable triggers for the action. 7.8.3 Discussion In browsers whose object models support access to all element types (IE 4 and later or NN 6 and later), you can control the values of the cols and rows attributes of the frameset element via properties of the same names. The frameResize.js code library shown in Example 7-1 adjusts only the cols property of a typical two-column frameset. In this case, the width of the lefthand frame is set to zero, effectively hiding the frame from view. Example 7-1. Frame-resizing functions in frameResize.js // global to save previous setting var origCols; // resize lefthand franem function resizeLeftFrame(left) { var frameset = document.getElementById("masterFrameset"); origCols = frameset.cols; frameset.cols = left + ", *"; } function restoreFrame( ) { document.getElementById("masterFrameset").cols = origCols; origCols = null; } function toggleFrame( ) { if (origCols) { restoreFrame( ); } else { resizeLeftFrame(0); } } The code begins by declaring a global variable, origCols, that preserves the original cols attribute setting so it can be used later to restore the previous setting. Next, the resizeLeft( ) function is wired to apply a numeric value (arriving as an argument) to the first of two values of the frameset's cols property. Before doing so, the function stores a copy of the property value in the origCols variable. To return the frameset to its previous state, invoke the restoreFrame( ) function. These two functions are controlled through a master function, toggleFrame( ), invoked from user interface elements located in one or more of the visible frames. A logical user interface idea behind the frame toggle is to change the text or image signifying the purpose of the click action. For example, when the frame is showing, the label says something like "Hide Navbar"; when the bar is invisible, the label says "Show Navbar." You can do this quite easily on IE browsers, but not Netscape 6 or 7. Here's how to implement the toggleFrame( ) function to work this IE magic: function toggleFrame(elem) { if (origCols) { restoreFrame( ); elem.innerHTML = "<<Hide Navbar"; } else { resizeLeftFrame(0); elem.innerHTML = "Show Navbar>>"; } } The button in the frame is: <<Hide Navbar Figure 7-1 shows the two states of the button with respect to the frameset. The reason Netscape 6 and 7 won't let you handle the UI portion so elegantly is that when the frameset resizes itself to the new specifications, the browser automatically reloads the pages (needlessly, in my opinion). Thus, the default value for the UI element reverts to the value as delivered from the server. To counteract the problem�and provide a cross-browser service to all users�you can preserve the state of the frame's visibility in a cookie, and use the cookie to determine the text of the button (or image URL if you prefer) each time the frame loads. This added work lets the user's preference persist from session to session, thus enhancing the visitor's enjoyment of the site. Coding for this enhancement has to work around an unfriendly bug in some versions of IE, where an event handler invoking a function in the parent provides incorrect information about the actual target of the event. The solution requires a little bit of indirection, but the result works in all browsers that support W3C DOM syntax. The page in the frame that always stays visible defines a placeholder span element whose content is filled only if the browser is of a required minimum scriptability: Scripts include a linked cookies.js library (see Recipe 1.9): An onload event handler in that frame's page initiates the creation of the UI element for the frame state toggle: function setToggleUI( ) { var label = ""; } var newElem = document.createElement("button"); newElem.onclick = initiateToggle; var newText = document.createTextNode(label); newElem.appendChild(newText); document.getElementById("togglePlaceholder").appendChild(newElem); } } Notice that the button element's onclick event handler invokes the local function initiateToggle( ) in the same frame. This indirection allows the accurate event target to be reported in IE. That function reads the target and sends it along to the revised toggleFrame( ) function in the parent frame: function initiateToggle(evt) { evt = (evt) ? evt : event; var elem = (evt.target) ? evt.target : evt.srcElement; if (elem.nodeType = = 3) { elem = elem.parentNode; } parent.toggleFrame(elem); } In the parent frame (which must also link in the cookies.js library from Recipe 1.9), the toggleFrame( ) function now not only adjusts the cols setting of the frameset, but it also saves the current state value to a cookie and sets the button's label to the new version (since IE doesn't reload the frameset, it needs the new values directly): function toggleFrame(elem) { if (origCols) { elem.firstChild.nodeValue = ""; setCookie("frameHidden", "true", getExpDate(180, 0, 0)); resizeLeftFrame(0); } } All other functions in the parent stay the same as in the original solution. That Netscape 6 and 7 (and perhaps later versions, as well) reload the frameset each time you adjust a frame's dimensions argues against implementing gradually expanding or contracting frames. Flicker from each reload may drive users crazy. 7.8.4 See Also Recipe 7.7 for reading the current pixel dimensions of a frame; Recipe 1.9 for details on the cookies.js library. Page 8 NN 6, IE 4(Win)/5(Mac) 7.7.1 Problem You want to know the precise pixel dimensions of a frame after the frameset is rendered. 7.7.2 Solution Because it is not uncommon to specify a percentage or * wildcard character for one or more frame size specifications, the actual rendered size is unknown until the browser completes all of its calculations for rendering. A script running in one of the frames can access the interior dimensions of any sibling frame (and its own frame) with the following function: function getFrameSize(frameID) { var result = {height:0, width:0}; if (document.getElementById) { var frame = parent.document.getElementById(frameID); if (frame.scrollWidth) { result.height = frame.scrollHeight; result.width = frame.scrollWidth; } } return result; } The function returns an object with height and width properties containing the rendered pixel measures of the frame whose ID is passed as a parameter. You get accurate readings in IE 4 or later for Windows and Netscape 7 or later. IE for Macintosh through Version 5.x provides accurate readings when the frame in question does not have scrollbars. When scrollbars are present, the returned sizes are reduced by the thickness of the scrollbars. Unfortunately, no property reveals whether a scrollbar is visible in the frame if the frame's scroll attribute defaults to the auto setting. 7.7.3 Discussion The keys to making the function work include referencing the parent (or top) of the individual frames and the actual frame element as opposed to the frame-as-window object. Thus, the more common parent.frameName or parent.frames["frameName"] types of reference will not suffice. By referencing the frame element directly, you get the browser to report the rendered size of the element, irrespective of the content of the frame. You can also install this function inside the frameset document's script. But you must then remove the reference to the parent in the fourth line of the function. Because the script already operates in the parent, its own element collection includes the frame elements. You can install this function in the frameset, yet still invoke it from one of the frames, as in: var frameSize = parent.getFrameSize("myFrame4"); The parent window is part of the equation in one form or another. 7.7.4 See Also Recipe 7.8 for resizing a frame under script control. Page 9 NN 2, IE 3 7.6.1 Problem You want a page bookmarked from a frame within a frameset to load within its frameset. 7.6.2 Solution For a frameset that always consists of the same framed documents, include a script patterned on the one below in each frame's document. This solution includes a test for a peculiarity of the Netscape Navigator 4 browser described in the Discussion section: var isNav4 = (navigator.appName = = "Netscape" && parseInt(navigator.appVersion) = = 4); if (top.location.href = = self.location.href) { if (isNav4) { if (window.innerWidth != 0) { top.location.href="frameset12.html"; } } else { top.location.href="frameset12.html"; } } If a user has bookmarked a page with the previous script in it, the page will begin to load, but will then load the frameset that normally houses the page. 7.6.3 Discussion The Navigator 4 issue affects the Windows version in particular. In that browser, when a user prints a frame, the page is automatically loaded into a temporary window of zero width. Scripts run when the page loads into that invisible window, so it's important to bypass the frameset-loading routine when the page is being printed or else you'll land in an infinite loop. If Navigator 4 is not among your target audience, you can simplify the script to just one branch: if (top.location.href = = self.location.href) { top.location.href="frameset12.html"; } This script compares the URLs of the windows referred to as top and self. If they are the same, it means that the document is loading as the only one in the browser window. You'll have to tailor the URL assignment for each document containing this script so that the desired frameset loads. For a more fluid frameset, where one or more frames changes in response to user action, you need more intelligence built into the scripts and documents. At the core, each page's document needs a script not only to load the frameset, but to convey the URL of the page that needs loading into one of the frames. For example, if a frameset contains a fixed navigation frame through a product catalog, and the content frame's document changes with each menu choice, you want a bookmark to one of the product frames to load the frameset with that product showing. To make this intelligent frame loading proceed correctly requires scripts in each bookmarkable page and one in the frameset. Each page sends information about itself by way of a search string appended to the frameset page's URL; the frameset's script receives that information and loads the content frame accordingly. Let's start with a content document. The script gets all of its information from the page itself, so you can write this script once as an external library and link it into each content page. Here is the library script, which runs as the page loads: var isNav4 = (navigator.appName = = "Netscape" && parseInt(navigator.appVersion) = = 4); if (parent = = window) { // Don't do anything if NN4 is printing frame if (!isNav4 || (isNav4 && window.innerWidth != 0)) { if (location.replace) { // Use replace( ), if available, to keep current page out of history location.replace("masterFrameset.html?content=" + escape(location.href)); } else { location.href="masterFrameset.html?content=" + escape(location.href); } } } The current page's URL is attached to the URL of the frameset in the form of a name/value pair, just like a typical form submission search string. All browsers except some early ones apply the location.replace( ) method so that the current page won't be added to the browser's history. If the page were to be part of the history, a user clicking the Back button could get into an infinite backward loop because the button would never be able to go any earlier than this page. Code for the frameset consists of two functions, one of which is triggered by the onload event handler of the tag. The key supporting function converts the search string from the URL into an object whose property name is the name part of the name/value pair passed in the search string. The function also accommodates multiple name/value pairs embedded in the search string in case you wish to pass additional information to the frameset: function getSearchData( ) { var results = new Object( ); if (location.search.substr) { var input = unescape(location.search.substr(1)); if (input) { var srchArray = input.split("&"); var tempArray = new Array( ); for (var i = 0; i < srchArray.length; i++) { tempArray = srchArray[i].split("="); results[tempArray[0]] = tempArray[1]; } } return results; } The onload event handler of the frameset invokes the following loadFrame( ) function, which reads the search string data and applies the URL to the content frame: function loadFrame( ) { if (location.search) { var srchArray = getSearchData( ); if (srchArray["content"]) { self.content.location.href = srchArray["content"]; } } } This example uses the same frame name (content) as the property name in the name/value pair of the search string. Theoretically, you could generalize the loadFrame( ) function to assign the URL to the frame whose name arrives in the name/value pair. That design choice, however, places the burden of knowing the destination frame at the individual frame level. By leaving the specification to the frameset script, the destination frame's name is kept within the context of the affected frameset. Assuming that the framesetting document contains a default URL for the content frameset (i.e., for when the visitor first arrives to the frameset), that default page stays in place if no search string containing a content property name arrives with the frameset URL. If a search string accompanies the URL, the default page appears momentarily, until the passed URL is loaded into the frame. 7.6.4 See Also Recipe 7.5 for a script that keeps your site from being encapsulated in someone else's frameset; Recipe 10.6 for more examples of passing data between pages via URLs. Page 10 NN 2, IE 3 7.5.1 Problem You want to prevent your site from appearing within another site's frameset. 7.5.2 Solution Include the following script statements at the very top of a script appearing in the head portion of the document: if (top != self) { top.location.href = location.href; } If you are using a frameset, include this script only in the framesetting document's script, and not in the documents that appear in the frames. 7.5.3 Discussion The act of framing someone else's site is less prevalent than it was some years ago, but it can still happen. Sometimes it occurs innocently enough, when another site includes a pointer to your site but the link is part of a frameset navigation system, where all link destinations are loaded into a content frame of the site. Whether you are concerned that your site appears in a frame of someone else's site is a personal issue. A corporate site usually desires control of the user's experience at the site, and doesn't wish to have someone else's banner advertising appear in the same browser window as its own pages. Also, being in someone else's frameset makes it less likely that the visitor will correctly bookmark your site. Not all users�especially non-technical casual web surfers�know that the browser's contextual menu includes an option to bookmark just one frame. The script in this recipe compares the current window object against whatever window object is reflected by the top window reference. A window object has four different ways to be referenced: window, self, parent, and top. There is no difference between window and self: they always refer to the current window regardless of the window's relationship among frames and framesets. Sometimes, as in this case, using self is more descriptive when you read the script. When the browser window contains no framesets, the current window object is also the parent and top object. If that condition isn't met when the page with the script shown in the Solution loads, the frameset gets replaced by the page running the script. Because the unknown outer frame is undoubtedly originating from another domain and server, you cannot dig too deeply into the details of the top window, such as its URL. Security restrictions prevent that. But comparing the window object references does not violate the same origin policy. 7.5.4 See Also Recipe 7.6 for a way to guarantee that a URL to one of your framed documents loads in its frameset. Page 11 NN 2, IE 3 7.4.1 Problem You want a script statement in one frame to replace the frameset with a new document, such as removing your frameset to make way for an entirely different web site. 7.4.2 Solution Assign the URL to the topmost parent of the frame hierarchy: top.location.href="newPage.html"; This is the scripted equivalent of the following hyperlink specification: ... Use the scripted version only if you are performing some other script activity leading up to the replacement, or you need your page to validate to the XHTML strict DTD, which forbids the target attribute. 7.4.3 Discussion You could reference the parent frameset instead of the top in a simple frameset. In this instance, a reference to the top and parent framesets points to the same object. But if you specify the topmost frameset, you are guaranteed removal of any vestiges of earlier framesets, especially if the script is potentially running within a nested frameset. That includes a frameset from an origin other than your own domain and server. 7.4.4 See Also Recipe 7.5 for how to use this technique to prevent your site from being displayed in another site's frameset. Page 12 NN 2, IE 3 7.3.1 Problem You want a single button or hyperlink to change the content of two or more frames of the frameset. 7.3.2 Solution Scripting is required for two frame changes at once. It is best to define a generalized function that performs the navigation, and invoke that function from an onclick event handler of a button or hyperlink. A function format that works for both interface elements is like the following: function loadFrames(url1, url2) { parent.otherFrameName.location.href = url1; location.href = url2; return false; } The hyperlink tag that invokes this function looks as follows: ... A default navigation path for the link is provided for those with scripting disabled. 7.3.3 Discussion There are numerous variations on the solution script. Your choice depends on your design and the audience for the pages. Let's examine some other scenarios and alternate approaches. As indicated in Recipe 5.10, your site design can allow for nonscripted access to be controlled strictly by the standard hyperlink href and target attributes, but a script-enhanced presentation takes advantage of the onclick event handler of the link to supplement or replace the default hyperlink action. You can even go so far as to have the default action navigate to one part of your web site, while the scripted action goes down an entirely different path suited to scriptable browsers. The example shown in the Solution is tailored to a link that changes documents in both the current frame and one other. You aren't limited to this combination. If you'd rather keep the current frame intact (perhaps it is a static navigation bar), but multiple other frames are to be updated with each navigation bar click, change the references in the loadFrames( ) function to point to the desired frames in the order the URLs arrive from the event handler calls. When using hyperlinks as the user interface element, however, be sure to use the technique shown in the example so that the onclick event handler ultimately evaluates to return false to prevent the default link action from operating. If access by nonscriptable browsers is a significant issue for your design, and you have a complex frameset consisting of several frames, you can still offer the equivalent of changing multiple frames, but do it for scriptable browsers more quickly. For nonscriptable browser users, you have to define a different framesetting document for each combination of frame content you offer from your navigation menu. Assign the framesetting document to the href attribute of a link, with the target pointing to the special _top window. Continue to use the loadFrames( ) function for the onclick event handler of the link: ... The advantage for the scriptable browser user is that the frames that don't have to change stay right where they are, and the browser doesn't have to compare the cache for those frames against the server's document. Thus, navigation is faster for scriptable browsers, but just as complete for nonscriptable browsers. This is the ideal value-added proposition that DHTML brings to a web page. 7.3.4 See Also Recipe 7.2 for changing the content of one sibling frame. Page 13 NN 2, IE 3 7.2.1 Problem You want a script in one frame to change the document in another frame. 7.2.2 Solution Assign the URL to the location.href property of the sibling frame with the following syntax: parent.otherFrameName.location.href="newPage.html"; Replace otherFrameName with the name assigned to the name attribute of the tag designated for that other frame in the framesetting document. 7.2.3 Discussion You don't need JavaScript at all to load a page into another frame if the user is clicking on a hyperlink. Instead, assign the frame's name to the target attribute of the tag, and let the href assignment handle the navigation part: Product Catalog On the other hand, if you want your pages to validate under strict XHTML, the target attribute is not allowed for hyperlink elements (your framesetting document also wouldn't validate under strict XHTML, but you might be satisfied with letting that document validate under the frameset version of XHTML 1.0). If you need a target attribute but can't use it due to validation conflicts, you can use scripts to fill in the target attribute values after the page loads. Assign an onload event handler to the page so that a function assigns frame names to the target properties of all links on the page. Thanks to the document.links collection found in every HTML page, a script can easily loop through them all and assign the values as needed. You can even segregate links that are to be loaded into another frame from those that are to replace the current frame: use the class attribute of the a elements to divide the elements into two classes. The following function (for IE 4 or later and NN 6 or later) assigns the content frame target to those links whose class attributes are set to other: function setLinkTargets( ) { for (var i = 0; i < document.links.length; i++) { if (document.links[i].className = = "other") { document.links[i].target = "content"; } } } Bear in mind that client-side image map area elements are also counted among the document.links collection. Include them or not in this scripted assignment task, as your design requires, by choosing which area elements have their class attributes assigned in the HTML code. 7.2.4 See Also Recipe 7.3 to change multiple frames from one user action; Recipe 7.4 for replacing the current frameset with an entirely new document or frameset. Page 14 NN 2, IE 3 7.1.1 Problem You want a frameset definition to include a blank frame (as a clean slate, awaiting a menu selection), but without having to create a blank HTML document on the server. 7.1.2 Solution The following framesetting page demonstrates the technique of using a script-generated blank page in one of two frames: You can apply the javascript: pseudo-URL to the src attribute of any frame element. 7.1.3 Discussion As you can probably deduce from the example in the Solution, you can use JavaScript to create any HTML as the initial content of a frame. For example, if you wanted to use a special background color of the blank frame to be the same as your HTML pages in the frameset, you could include the bgcolor attribute of the tag inside the blankFrame( ) function: function blankFrame( ) { return ""; } Using the javascript: protocol with a src attribute is a somewhat controversial subject. On the one hand, it is the only backward-compatible way to let dynamic content fill an element that normally gets its content from a file or CGI process on the server. But this kind of URL fails if the user has JavaScript disabled or the browser doesn't support JavaScript. Most browsers with JavaScript disabled will simply leave the frame area blank, but the browser may be in an unstable state. Therefore, deploy this technique only if you know your audience has script-enabled browsers. Notice that the reference to the function points to the parent frame. This is required because the execution of the javascript: pseudo-URL occurs inside the context of the frame. In the frame's eyes, the function is located in the parent, and the reference must include that pointer. As an aside, a common mistake for scripting beginners is to replicate the javascript: URL in event handlers. I don't know where this came from, but it is wrong, redundant, and sometimes disastrous. The javascript: protocol belongs only where a URL is normally assigned. Thus, it is appropriate in assignments to src and href attributes in tags. Do not use it in event handlers. Period. 7.1.4 See Also Recipe 7.2 for modifying a frame's content after the frameset loads; Recipe 14.1 for creating dynamic content during the loading of a single page. Page 15 As with multiple windows, multiple frames are controversial among experienced web designers. Some love them, others refuse to use them. Dislike for framesets has a couple of origins. One dates back many years, when not all browsers supported them. Many veteran designers refused to accept framesets then and the prejudice continues. More recently, however, the pure and strict XHTML implementations omit frames from the document markup vocabulary. Forms and hyperlinks in validating documents cannot even include a target attribute that loads the result of a form submission or a linked document into another frame. But the frames concept is not disappearing into oblivion. The XHTML specification includes a frame-specific version, and future work at the W3C will likely provide a fresh, XML-based frame markup vocabulary (currently called XFrames). At the same time, virtually every graphical user interface browser in use today supports HTML frames, and will do so for a long time to come. By setting the frameset element's border attribute to zero to create a seamless space, users may not even be aware of your frame structure. Frames are especially useful in a few specific instances. The most common application is dividing a page into a large content frame and a smaller frame that acts as an index, table of contents, or site navigation menu. Such small frames might be along the left or right edge of the window, or sometimes as a horizontal slice at the top of the window. As the user scrolls and navigates content in the large frame, the smaller frame remains fixed and in position, ready for the next action. With the navigation frame remaining stable as the larger frame moves from page to page, the user does not have to wait for the navigation frame content to reload at every content page refresh. Another advantage to this relative stability is that you can use the framesetting document or other frame as a temporary repository of JavaScript data that persists while content pages change. 7.0.1 Frames as Window Objects Since the days of the earliest scriptable browsers, the browser's object model exposed frames to scripts, but not in the supplemental way that the latest W3C DOM does. Each frame in the original system (still very valid in today's browsers) is treated as a window. After all, a frame contains a document just like a regular browser window. If you can gain scriptable access to the frame, most window object properties and methods apply directly to the frame as well. The model for visualizing the relationships between frames in this manner uses a parent-child metaphor. The initial document that loads into the browser�the one containing the frameset element and all specifications for the frameset's makeup�is the parent of the frames containing documents that the user sees. Parent and individual child frames are all treated as window objects. A script running in the framesetting document can reference any of the child frames, and thus their documents, via the frames property of the parent window. The frames property contains a collection (array) of frame elements belonging to the parent: window.frames[i] window.frames["frameName"] window.frameName Two of the reference syntaxes rely on the name attribute of the frame elements being set. As for the numeric index, it is zero-based and follows source code order of the frame elements, even if the frames are nested deeply in framesets defined in the same top-level frameset document. In other words, regardless of the number of frameset elements defined in a framesetting document, there is only one parent, and as many child frames as there are frame elements defined in the document. A more complex relationship exists, however, if one of the documents assigned to a frame's src attribute is, itself, another frameset. A script in the top-level frameset accesses this kind of nested frame through the following hierarchy: window.frames["anotherFramesetName"].frames["nestedFrameName"] Scripts operating inside a frame can reference both the parent frameset as well as sibling frames, but references must follow the hierarchy rather strictly. The parent keyword is the gateway to the parent framesetting document. For example, if the framesetting document contains a global variable named allLoaded, a script in one of the frames can read that value this way: parent.allLoaded For a script to access one of its siblings, the reference must include a parent frameset that both siblings have in common. For example, consider the following simple frameset: A script in the navigation frame can access the content frame with any of the following references: parent.frames[1] parent.frames["content"] parent.content Thus, a script in the navigation frame can instruct the content frame to scroll to the top as follows: parent.frames["content"].scrollTo(0,0); If the document loaded into the content frame was, itself, a framesetting document, the reference lengthens to include the pathway to one of its nested frames: parent.frames["content"].frames["main"].scrollTo(0, 0); To simplify references between frames within deeply nested framesets, you can always begin a reference from the topmost frameset, and then work your way down to the desired frame. That's where the top keyword is most useful: top.frames["content"].frames["main"].scrollTo(0, 0); A script in the deeply nested main frame can gain ready access to the highest navigation frame as follows: top.frames["navigation"] Perhaps more important than all of these referencing scenarios is the concept that referencing frames as windows gives you immediate access to the document of that window. For example, if a script in one frame wants to read the value of a text box in a sibling frame, the following backward-compatible syntax uses all original JavaScript and DOM Level 0 conventions: var val = parent.frames["content"].document.entryForm.entryField.value; Don't forget to include the document reference after the frame reference (a common mistake). 7.0.2 Framesets and Frames as Elements In contrast to the frame-as-window scenario, the Internet Explorer and W3C object models allow scripts to reference the elements that create the framesets and frames. These objects have nothing to do with windows, per se, but everything to do with the ways the object models treat elements. Thus, these objects grant your scripts access to properties that mirror the tag attributes, such as a frameset's cols and rows properties and a frame's src and noResize properties. Access to these properties is handy when your scripts need to read or modify the attribute values. For example, Recipe 7.8 and Recipe 7.9 demonstrate how to adjust the dimensions of frames and even the column and row makeup of a frameset under script control. The question that arises, however, is how a script that has a reference to a frame element can reach the document inside the frame. For this, you need to access a special property of the frame element�more accurately, one of two possible properties, depending on the object model you are using. The IE model features a contentWindow property of a frame element, through which you can get to the document. For the W3C DOM, the contentDocument property references the document object inside the frame. Netscape 7 and later has implemented both, but if you need solid cross-browser support for IE 4 or later and Netscape 6 or later, use the following equalizer utility function: function getFrameDoc(frameElem) { var doc = (frameElem.contentDocument) ? frameElem.contentDocument : ((frameElem.contentWindow) ? frameElem.contentWindow.document : null); return doc; } 7.0.3 Frames and Events One common problem facing scripters who are new to frames is cross-frame scripts almost always rely on the other frame being loaded to operate correctly. If one frame loads quickly and references a form in a sibling frame but that form's document has not yet loaded, a script error greets the user. To compensate for this behavior, you must be mindful of the onload event handler characteristics for frames and framesets. Just like a window, each frame element can have an onload event handler. The onload event for a frame fires when the complete contents of that frame's document have reached the browser. The frameset element, too, receives an onload event, but only after all of the nested frames' documents have loaded (and each of their onload events have fired). Therefore, the only sure way to trigger functions that operate across frames is to do so via the frameset's onload event handler (which you can bind as an attribute to the actual tag). Don't confuse this behavior with the kind of event bubbling that graces the IE and W3C DOM event models. The frameset element's onload event handler fires only the first time the frameset and its child frames load. If the user or a script changes the URL of one of the frames, that frame's onload event fires, but the frameset's onload event does not fire again. If you assign an onload event handler to the tag in the framesetting document, it executes each time the content of that frame changes (but only in IE 5.5 or later for Windows and Netscape 7 or later). For older browsers, the onload event handler must be defined in the body of the loaded document). When documents in your framesets change a lot, and when they have substantial dependence on each other's scripting, you can also use another less elegant, but effective, technique to poll for the availability of another frame's document. Assuming that each document is served from the same domain and server (to satisfy the same-origin security policy), the onload event handler in each document sets a global Boolean variable, which acts as a flag for other frames' access. The variable is initialized at the top of the script as false, but the onload event handler sets it to true: var loaded = false; ... ... Any script that needs to access content or scripts in this document can check first for the value of the loaded variable, and check it every second or so until it is true or the number of permitted attempts reaches your maximum tolerance level: // count attempts to reach other frame var tries = 0; // the function that needs info from the other frame function someFunc( ) { if (parent.otherFrameName.loaded) { // OK, other frame is ready; use it in this branch tries = 0; // prepare for next access ... } else if (tries < 5) { tries++; // try again in 1 second setTimeout("someFunc( )", 1000); } else { tries = 0; alert("Sorry, we could not complete this task."); } } 7.0.4 Hidden Frames Occasionally, a site designer wants to use JavaScript to maintain an open connection with the server so that updated data from the server is reflected within a portion of the main page. While Internet Explorer's Data Binding technology is capable of doing this without any trickery, it is a Microsoft-only solution (with some additional restrictions for the Macintosh version of IE). For a cross-browser solution, other strategies are needed. One solution that works in many mainstream browsers is to embed a faceless Java applet that keeps its own socket open with the server. Two-way communication between the applet and the JavaScript in the HTML page (dubbed "LiveConnect" by Netscape's first implementation of this feature) allows a free flow of data swapping between the two environments. Successful deployment of this solution, however, requires that the user choose to include the Java runtime environment with the browser. An alternate solution is to use a hidden frame containing a document that periodically polls the server for updated data. Other than perhaps seeing some unexplained network activity, the user won't even know the unseen frame is refreshing itself every few seconds. The URL of the hidden frame could be to a CGI program running on the server. Such a CGI program then delivers a page with no renderable content, but with data embedded in the document in JavaScript or XML form (see Recipe 14.4 and Recipe 14.5 for further ideas about this data formatting). Additional scripts in the returned data, triggered by the onload event handler of the page, can copy data from the hidden frame to designated table cells or other HTML text nodes in the visible page. 7.0.5 Frame No-Nos Just as windows and window objects could expose users to unscrupulous sites if security precautions were not in place, frames could offer a similar range of holes. But those, too, are blocked. Because frames are not as rich as windows with respect to their impact on the browser application, the list of things you can't do with frames is much shorter than for windows. Regardless of your good intentions, you cannot do the following things with frames: Accessing document properties from other frames served by other domains It could be potentially nasty business if a script in one frame could look into another frame and retrieve its URL or any of the HTML content. Browsers implement what is known as a same-origin security policy, which means that a script in one frame (or window) cannot access critical details (URL, DOM structure, form control settings, text content) from another frame (or window) unless both pages are delivered by the same domain and server, and arrived via the same protocol (such as HTTP or HTTPS, but not both). Changing the content of the Address/Location text field The URL shown in the field is of the framesetting document. You cannot place, say, the main content frame's URL in the field. Setting a Favorites/Bookmarks entry to maintain the precise frameset composition If a user navigates through your framed web site, the browser's own bookmarking facilities will preserve either the frameset or (through a contextual menu) a single frame's document. See Recipe 7.6 to assist in reconstructing a frameset from one frame's bookmark. Page 16 Section 7.0. Introduction Section 7.1. Creating a Blank Frame in a New Frameset Section 7.2. Changing the Content of One Frame from Another Section 7.3. Changing the Content of Multiple Frames at Once Section 7.4. Replacing a Frameset with a Single Page Section 7.5. Avoiding Being "Framed" by Another Site Section 7.6. Assuring a Page Loads in its Frameset Section 7.7. Reading a Frame's Dimensions Section 7.8. Resizing Frames Section 7.9. Setting Frameset Specifications Dynamically Page 17 NN 6, IE 5 6.10.1 Problem You want to create the impression of a separate draggable window but without actually creating a new window. 6.10.2 Solution This solution consists of many individual files, including three .js JavaScript library files and several .css style sheet files. One of the library files, DHTML2API.js, is applied directly from Recipe 13.3; the other libraries, layerDialog.js and layerDialogDrag.js, which take care of creating the window and making it draggable, respectively, are shown in the Discussion. The .css files provide different looks for the window on different operating systems; they are also shown in the Discussion. The following skeletal HTML main page shows the JavaScript libraries as they are linked into the page, the makeup of the required elements (some div, span, and iframe elements) that comprise the pseudowindow, and a sample hyperlink element that invokes the openLayerDialog( ) function to display the modeless, draggable pseudowindow (in this example, a Preferences window): Main Application Page // function to run upon closing the dialog with "OK". function setPrefs( ) { // Statements here to apply choices from the dialog window } Preferences Titlebar Three key event initializations routines, initDHTMLAPI( ) (from DHTML2API.js), initDrag( ) (from layerDialogDrag.js), and initLayerDialog( ) (from layerDialog.js) are invoked by the onload event handler of the tag. 6.10.3 Discussion Specifying the functional characteristics of a positioned element acting as a simulated window gets a bit complicated if you wish to take into account the wide variety of looks and feels of popular operating systems. A window design tailored for Windows 98 won't look anything like a Windows XP window. Even in the Mac world, there is a traditional look to Mac OS 9 and the radically different Mac OS X. This recipe makes the following assumptions about the pseudowindow constructed from positioned elements: The window is a fixed size, governed by style sheets (the example shows 800x600) Titlebars and close boxes are customized for Windows 9x, Windows XP, Mac OS, and Mac OS X Other operating systems default to the Windows 9x look The content of the window can be loaded from any URL The window is draggable, behaving like a modeless dialog window The element-based pseudowindow is a draggable composite object. An outer wrapper div (pseudoWindow) is the overall container for the pseudowindow. Nested inside are two child elements. The first represents the pseudowindow's titlebar. Two child nodes of the titlebar div element are an image for the window's close box and a span for the window's text title. The second is an iframe element, which holds the content. At the core of the interactivity built into the pseudowindow are three JavaScript libraries: DHTML2API.js, layerDialog.js, and layerDialogDrag.js. Because the DHTML2API.js library (Recipe 13.3) contains numerous cross-browser functions for element positioning, the current recipe loads this library to support the positioning tasks required of the pseudowindow. Example 6-2 shows the code for the layerDialog.js library, which has two important jobs: linking in the OS-specific style sheets for the pseudowindow, and controlling the creation, initial position, and display of the pseudowindow. Example 6-2. layerDialog.js library // Help choose from four UI pseudo-window flavors function getCurrOSUI( ) { var ua = navigator.userAgent; if (ua.indexOf("Mac") != -1) { if (ua.indexOf("OS X") != -1 || ua.indexOf("MSIE 5.2") != -1) { return "macosX"; } else { return "macos9"; } } else if (ua.indexOf("Windows XP") != -1 || ua.indexOf("NT 5.1") != -1) { return "winxp"; } else if ((document.compatMode && document.compatMode != "BackComp") || (navigator.product && navigator.product = = "Gecko")) { // Win9x and CSS-compatible return "win9x"; } else { // default for Windows 9x in quirks mode, Unix/Linux, & unknowns return "win9xQ"; } } var currOS = getCurrOSUI( ); // Load OS-specific style sheet for pseudo dialog layer document.write(""); //***************************** // Begin Layer Dialog Code //******************************/ One object tracks the current pseudo-window layer. var dialogLayer = {layer:null, visible:false}; // Center a positionable element whose name is passed as // a parameter in the current window/frame, and show it function centerOnWindow(elemID) { // 'obj' is the positionable object var obj = getRawObject(elemID); // window scroll factors var scrollX = 0, scrollY = 0; if (document.body && typeof document.body.scrollTop != "undefined") { scrollX += document.body.scrollLeft; scrollY += document.body.scrollTop; if (document.body.parentNode && typeof document.body.parentNode.scrollTop != "undefined") { scrollX += document.body.parentNode.scrollLeft; scrollY += document.body.parentNode.scrollTop } } else if (typeof window.pageXOffset != "undefined") { scrollX += window.pageXOffset; scrollY += window.pageYOffset; } var x = Math.round((getInsideWindowWidth( )/2) - (getObjectWidth(obj)/2)) + scrollX; var y = Math.round((getInsideWindowHeight( )/2) - (getObjectHeight(obj)/2)) + scrollY; shiftTo(obj, x, y); } function initLayerDialog( ) { document.getElementById("closebox").src="closeBox_" + currOS + ".jpg"; dialogLayer.layer = document.getElementById("pseudoWindow"); } // Set up and display pseudo-window. // Parameters: // url -- URL of the page/frameset to be loaded into iframe // returnFunc -- reference to the function (on this page) // that is to act on the data returned from the dialog // args -- [optional] any data you need to pass to the dialog function openLayerDialog(url, title, returnFunc, args) { if (!dialogLayer.visible) { // Initialize properties of the modal dialog object. dialogLayer.url = url; dialogLayer.title = title; dialogLayer.returnFunc = returnFunc; dialogLayer.args = args; dialogLayer.returnedValue = ""; // Load URL document.getElementById("contentFrame").src = url; // Set title of "window" document.getElementById("barTitle").firstChild.nodeValue = title; // Center "window" in browser window or frame dialogLayer.layer.style.visibility = "hidden"; dialogLayer.layer.style.display = "block" centerOnWindow("pseudoWindow"); // Show it and set visibility flag dialogLayer.layer.style.visibility = "visible" dialogLayer.visible = true; } } function closeLayerDialog( ) { dialogLayer.layer.style.display = "none" dialogLayer.visible = false; } //************************** // End Layer Dialog Code //****************************/ The layerDialog.js library begins by loading the external style sheet file matching the user's operating system. The getCurrOSUI( ) function uses browser operating system detection to determine which of the five supported styles applies to the current browser. Then the function dynamically writes the tag with the desired URL. The code for the five .css files can be found in Examples 6-4 through 6-8 later in this recipe. The library continues by defining a global variable that acts as an abstract object holding various pieces of information about the actual pseudowindow. A utility function adapted from Recipe 13.7 (centerOnWindow( )) centers the layer in the current browser window when asked to by the function that prepares the window for display. This application of the recipe, however, removes the final show( ) function call (to the DHTML API) because another function of the library controls the pseudowindow's visibility. A brief initialization routine in initLayerDialog( ) runs just after the page loads so that the correct close box art is downloaded at the outset. The dialogLayer global object also is assigned a reference to the layer for a shortcut reference in functions that operate on the layer. The main function of this library, openLayerDialog( ), is invoked by your scripts when they need to display the pseudowindow. This function shares many arguments with the functions in Recipe 6.9 that are used for simulating a modal dialog. However, the dimensions of the window are not needed here, since the pseudowindow is a fixed size. One final function, closeLayerDialog( ), is invoked when the user clicks on the pseudowindow's close box. If you're wondering why the openLayerDialog( ) function includes code that modifies both the style.display and style.visibility properties to show the layer, it is a result of a rendering bug in IE 6 for Windows. Using only the style.visibility property to hide the pseudowindow can leave the rectangular area of the iframe completely blank (white) after the layer hides itself. Using the style.display settings takes care of the bug. But this introduces a different problem with positioning the layer prior to showing it. The odd combination of turning the layer's visibility to hidden and the display to block before centering the window does the trick. Then it's safe to use the visibility property to present the layer to the user. Because this pseudowindow does not pick up the title from the document loaded into the iframe (security restrictions prevent such actions if the pseudowindow content is served from a different domain and server), you must pass the desired window titlebar text as the parameter to the openLayerDialog( ) function. The third parameter is a reference to a function that you want to invoke when the user clicks on an OK or Apply button in the window�very much like the function passed to the simulated dialog window in Recipe 6.9. This reference is also available as a property of the global dialogLayer object. The same is true for the fourth parameter of openLayerDialog( ), which you can use to pass data to the pseudowindow. Be aware, however, that the document in the pseudowindow's iframe won't be able to access the dialogLayer object if the content arrives from another domain and server. To display a new document in the pseudowindow without passing any function reference or arguments, supply a null parameter, as in: openLayerDialog("prefs.html", "User Preferences", null, null); Example 6-3 shows the layerDialogDrag.js library, which provides support for dragging the pseudowindow around by the element standing in for the titlebar. Example 6-3. layerDialogDrag.js library // Global holds reference to selected element var selectedObj; // Globals hold location of click relative to element var offsetX, offsetY; // Set global reference to element being engaged and dragged function setSelectedElem(evt) { var target = (evt.target) ? evt.target : evt.srcElement; var divID = (target.id = = "titlebar") ? "pseudoWindow" : ""; if (divID) { if (document.layers) { selectedObj = document.layers[divID]; } else if (document.all) { selectedObj = document.all(divID); } else if (document.getElementById) { selectedObj = document.getElementById(divID); } setZIndex(selectedObj, 100); return; } selectedObj = null; return; } // Turn selected element on function engage(evt) { evt = (evt) ? evt : event; setSelectedElem(evt); if (selectedObj) { if (document.body && document.body.setCapture) { // engage event capture in IE/Win document.body.setCapture(); } if (evt.pageX) { offsetX = evt.pageX - ((typeof selectedObj.offsetLeft != "undefined") ? selectedObj.offsetLeft : selectedObj.left); offsetY = evt.pageY - ((selectedObj.offsetTop) ? selectedObj.offsetTop : selectedObj.top); } else if (typeof evt.clientX != "undefined") { offsetX = evt.clientX - ((selectedObj.offsetLeft) ? selectedObj.offsetLeft : 0); offsetY = evt.clientY - ((selectedObj.offsetTop) ? selectedObj.offsetTop : 0); } else if (evt.offsetX || evt/offsetY) { offsetX = evt.offsetX - ((evt.offsetX < -2) ? 0 : document.body.scrollLeft); offsetX -= (document.body.parentElement && document.body.parentElement.scrollLeft) ? document.body.parentElement.scrollLeft : 0 offsetY = evt.offsetY - ((evt.offsetY < -2) ? 0 : document.body.scrollTop); offsetY -= (document.body.parentElement && document.body.parentElement.scrollTop) ? document.body.parentElement.scrollTop : 0 } evt.cancelBubble = true; return false; } } // Drag an element function dragIt(evt) { evt = (evt) ? evt : event; if (selectedObj) { if (evt.pageX) { shiftTo(selectedObj, (evt.pageX - offsetX), (evt.pageY - offsetY)); } else if (evt.clientX || evt.clientY) { shiftTo(selectedObj, (evt.clientX - offsetX), (evt.clientY - offsetY)); } evt.cancelBubble = true; return false; } } // Turn selected element off function release(evt) { if (selectedObj) { setZIndex(selectedObj, 0) if (document.body && document.body.releaseCapture) { // stop event capture in IE/Win document.body.releaseCapture(); } selectedObj = null; } } function blockEvents(evt) { evt = (evt) ? evt : event; evt.cancelBubble = true; return false; } // Assign event handlers used by both Navigator and IE function initDrag( ) { if (document.layers) { // turn on event capture for these events in NN4 event model document.captureEvents(Event.MOUSEDOWN | Event.MOUSEMOVE | Event.MOUSEUP); return; } else if (document.body & document.body.addEventListener) { // turn on event capture for these events in W3C DOM event model document.addEventListener("mousedown", engage, true); document.addEventListener("mousemove", dragIt, true); document.addEventListener("mouseup", release, true); return; } document.onmousedown = engage; document.onmousemove = dragIt; document.onmouseup = release; return; } The layerDialogDrag.js library is identical to the element-dragging library from Recipe 13.11, with one small modification. The setSelectedElement( ) function needs to acknowledge events only from the titlebar layer, but it must set the draggable layer to be the outer pseudoWindow layer. Thus, when the user drags the titlebar, the entire pseudowindow div element moves. The rest of Recipe 13.11 works as-is in this library, including the event handler definitions. You may experience a cosmetic annoyance in some circumstances, however. If the content of the pseudowindow has form controls, browsers running on slow computers don't refresh the screen promptly, leading to temporary ghosts while the pseudowindow is dragged. You'll have to evaluate how troubling this might be to your users based on the types of pages you load into the iframe. Let's now come back to the issue of style sheets and their influence on the pseudo-window elements. When style sheets are applied to the basic HTML of the pseudoWindow div element, you get a range of looks, as shown in Figure 6-1. One of the stated goals of this application was to have the pseudowindow blend into the operating system look and feel as much as possible. It's no longer a simple task, thanks to radically revamped user interfaces for Windows XP and Mac OS X. The example shown in Figure 6-1 tries to match the look and feel of four different operating system versions. To that end, a set of five separate external style sheets are created to handle the specific art files and sizes to come close to simulating the native look and feel of the host operating system. This assumes, of course, that you have available to you the art files for titlebar backgrounds (just small vertical slices needed because the backgrounds repeat to fit the space) as well as close boxes. So you can compare the subtle differences needed between the versions, we'll begin with dialogLayer_win9xQ.css (shown in Example 6-4), which is the style sheet used for backward-compatible IE versions (IE 5, 5.5, and 6 running in quirks mode). Example 6-4. dialogLayer_win9xQ.css for backward-compatible IE/Windows versions #pseudoWindow {position:absolute; top:0px; left:0px; width:600px; height:502px; border:2px solid black; background-color:#ffffff; border-top:3px solid #cccccc; border-left:3px solid #cccccc; border-right:3px solid #666666; border-bottom:3px solid #666666; display:none } #titlebar {position:absolute; top:0px; left:0px; height:16px; width:596px; background-image:url(titlebar_win9x.jpg); color:#ffffff; border-bottom:2px solid #666666; font-family:Tahoma; font-size:8pt; font-weight:bold; padding:2px; text-align:left } #closebox {position:absolute; right:0px; top:1px } #barTitle {padding-left:3px} #contentFrame{position:absolute; top:19px; left:0px; height:477px; width:594px; background-color:#ffffff; margin-left:0px; margin-top:0px; overflow:visible } Only minor differences in dimensions accrue to the CSS and Windows 9x-compatible version of the style sheet: dialogLayer_win9x.css (shown in Example 6-5). This variation is needed to account for the different ways that CSS-compatible browsers measure element widths when borders, margins, and padding are involved. The titlebar background art file is the same for both Windows 9x versions, as is the entire look of the pseudowindow. Example 6-5. dialogLayer_win9x.css CSS-compatible stylesheet for Windows 9x #pseudoWindow {position:absolute; top:0px; left:0px; width:600px; height:502px; border:2px solid black; background-color:#ffffff; border-top:3px solid #cccccc; border-left:3px solid #cccccc; border-right:3px solid #666666; border-bottom:3px solid #666666; display:none } #titlebar {position:absolute; top:0px; left:0px; height:16px; width:596px; background-image:url(titlebar_win9x.jpg); color:#ffffff; border-bottom:2px solid #666666; font-family:Tahoma; font-size:8pt; font-weight:bold; padding:2px; text-align:left } #closebox {position:absolute; right:0px; top:1px } #barTitle {padding-left:3px} #contentFrame{position:absolute; top:22px; left:0px; height:480px; width:600px; background-color:#ffffff; margin-left:0px; margin-top:0px; overflow:visible } The Windows XP version, dialogLayer_winxp.css (shown in Example 6-6), adjusts its dimensions to accommodate a thicker titlebar. Example 6-6. dialogLayer_winxp.css for browsers running in Windows XP #pseudoWindow {position:absolute; top:0px; left:0px; width:600px; height:502px; border:2px solid black; background-color:#ffffff; border-top:3px solid #cccccc; border-left:3px solid #cccccc; border-right:3px solid #666666; border-bottom:3px solid #666666; display:none } #titlebar {position:absolute; top:0px; left:0px; height:26px; width:596px; background-image:url(titlebar_winxp.jpg); color:#ffffff; border-bottom:2px solid #666666; font-family:Tahoma; font-size:10pt; font-weight:bold; padding:2px; text-align:left } #closebox {position:absolute; right:1px; top:1px } #barTitle {padding-left:3px} #contentFrame{position:absolute; top:30px; left:0px; height:472px; width:600px; background-color:#ffffff; margin-left:0px; margin-top:0px; overflow:visible } For the Macintosh version prior to Mac OS X, dialogLayer_macos9.css (shown in Example 6-7), the close box is on the left side of the titlebar, and minor dimensional differences are needed for the user interface elements. Example 6-7. dialogLayer_macos9.css for Macintosh browsers in OS 9 and earlier #pseudoWindow {position:absolute; top:0px; left:0px; width:600px; height:502px; border:2px solid black; background-color:#ffffff; border-top:3px solid #cccccc; border-left:3px solid #cccccc; border-right:3px solid #666666; border-bottom:3px solid #666666; display:none } #titlebar {position:absolute; top:0px; left:0px; height:16px; width:596px; background-image:url(titlebar_macos9.jpg); color:#000000; border-bottom:2px solid #666666; font-family:Charcoal; font-size:9pt; font-weight:normal; padding:2px; text-align:center } #closebox {position:absolute; left:0px; top:1px; padding-right:3px } #barTitle {padding-right:6px; background-color:#cccccc; padding-left:6px } #contentFrame{position:absolute; top:22px; left:0px; height:480px; width:600px; background-color:#ffffff; margin-left:0px; margin-top:0px; overflow:visible } The Mac OS X version, dialogLayer_macosX.css (shown in Example 6-8), has its own dimensional details. Example 6-8. dialogLayer_macosX.css for Macintosh browsers in Mac OS X #pseudoWindow {position:absolute; top:0px; left:0px; width:600px; height:502px; border:2px solid black; background-color:#ffffff; border-top:3px solid #cccccc; border-left:3px solid #cccccc; border-right:3px solid #666666; border-bottom:3px solid #666666; display:none } #titlebar {position:absolute; top:0px; left:0px; height:16px; width:596px; background-image:url(titlebar_macosX.jpg); color:#000000; border-bottom:2px solid #666666; font-family:Charcoal; font-size:9pt; font-weight:normal; padding:2px; text-align:center } #closebox {position:absolute; left:0px; top:0px; padding-left:5px } #barTitle {padding-right:6px background-color:transparent padding-left:6px } #contentFrame{position:absolute; top:22px; left:0px; height:480px; width:600px; background-color:#ffffff; margin-left:0px; margin-top:0px; overflow:visible } Perhaps the most significant advantage of using a positioned element as a pseudowindow is that no extraneous pop-up windows are spawned. Not only do some pop-up ad blockers prevent window.open( ) from working for some users, but multiple windows can get lost behind others, causing confusion among casual web surfers. Another, more philosophical, advantage is that the W3C DOM Level 2 pays only slight recognition to the window as an object. XHTML specifications recommend against the usage of multiple frames or windows (target attributes of form or a elements are not valid in strict XHTML 1.0, for example). Keeping everything in one browser window appeals to one-window designers. At the same time, however, using a layer to simulate a window presents disadvantages over and above the dragging issue noted earlier. The foremost concern is that a layer is confined to the window or frame boundaries in which it resides. Unlike a true window, the user cannot drag the layer to a position such that it extends beyond the main window or frame edges. If the main window's content is scrollable, the layer moves with the page when the user scrolls the main window while the pseudowindow is visible. Although the example shown here is for a fixed-size pseudowindow, you could expand upon the existing code to let the openLayerDialog( ) function receive and operate on two more parameters that specify the layer size. This gets more complex in a version that needs to support the same range of operating-system user interfaces, as this example does. If you compare the style sheet settings that control div element widths and some div heights and top measures, you can see that you must account for a wide variety of tweaks for a number of UIs. For example, the height, width, and top style sheet properties for the iframe vary between the CSS- and backward-compatible versions of the Windows 9x specifications. If you are prepared to uncover the factors affecting various elements in this pseudowindow, as well as

apply them to a fresh window size passed as parameters to openLayerDialog( ), you can make the function more malleable than shown here. The ideal scenario is deployment on an intranet where the browsers and operating systems that need supporting are strictly limited to a tolerable handful (or one!). Another expansion on the pseudowindow is to fashion the equivalent of a modal window, where the user is blocked from clicking on underlying links or form controls. You can accomplish this by wrapping the current pseudoWindow div inside yet another div whose background is a transparent image. The tricky part is sizing the outer wrapper to the dimensions of the document so that it doesn't extend to such an arbitrarily large size that the browser window's scrollbars let users scroll the page into blank space. You must then position the pseudoWindow div within the outer wrapper, taking the page scrolling into account. Trap all events in the outer div, and assign the dragging event handlers to that div as well (instead of the base document). The more browser types you want to support for this kind of feature, the greater the challenge. But it's doable if you are persistent and patient. 6.10.4 See Also Recipe 6.8 for proprietary IE modal and modeless dialog windows; Recipe 6.9 for a cross-browser simulated modal window using a subwindow; Recipe 11.5 for importing OS-specific style sheets; Recipe 13.3 for details of the DHTML API library; Recipe 13.7 for centering an element in a window; Recipe 13.11 for creating a draggable element; Recipe 14.14 for dynamically replacing a portion of body content. Page 18 NN 4, IE 4 6.9.1 Problem You want to present a consistent modal dialog on multiple browsers. 6.9.2 Solution Although IE provides the showModalDialog( ) method, no other browser supports it. This recipe uses a browser subwindow to simulate the behavior of a modal dialog box. It operates in IE 4 or later, Navigator 4 or later, and Opera 6 or later. Note that this is a simulation of true modality. Due to some odd behavior in IE for Windows with respect to disabling hyperlinks in the main window, a determined user can bypass the modality of this solution. For casual users, however, the window behaves much like a modal dialog box. Assemble your main HTML page around the simModal.js script library described in the Discussion. This library works by disabling form controls and links in the main page after the modal dialog is displayed and making sure the dialog keeps the focus, so that the user is forced to deal with the dialog. After the dialog is dismissed, the form controls and links are enabled again. The following skeletal HTML main page shows the event handler additions that the simModal.js library relies upon, and a demonstration of how to invoke the function that displays a simulated modal window (in this example, a Preferences window): Main Application Page // function to run upon closing the dialog with "OK". function setPrefs( ) { // Statements here to apply choices from the dialog window } Preferences Add the onclick and onfocus event handlers to the tag as shown. Those event handlers invoke the checkModal( ) event handler function defined in the external library to make sure the dialog window keeps the focus. Call the openSimDialog( ) function to display the window, passing the URL of the page to load into the dialog window, the window's width and height (in pixels), and a reference to a function in the main page that the modal window invokes when the window closes (setPrefs( ) in this case). In the dialog window's page, add the closeme( ), handleOK( ), and handleCancel( ) functions shown in the following extract to take care of the actions from the dialog window's Cancel and OK buttons. The onload and onunload event handlers of the tag trigger essential event-blocking services controlled by the blockEvents( ) and unblockEvents( ) event handlers in the simModal.js library. Preferences // close the dialog function closeme( ) { window.close( ); } // handle click of OK button function handleOK( ) { if (opener && !opener.closed && opener.dialogWin) { opener.dialogWin.returnFunc( ); } else { alert("You have closed the main window.No action will be taken on the " + "choices in this dialog box."); } closeme( ); return false; } // handle click of Cancel button function handleCancel( ) { closeme( ); return false; } If the dialog window contains a frameset (where the Cancel and OK buttons are in one of the frames), locate the onload and onunload event handlers in the tag. Keep the three functions in the framesetting document, and have the onclick event handlers of the buttons reference parent.handleCancel( ) and parent.handleOK( ). 6.9.3 Discussion Example 6-1 shows the entire simModal.js library, which you link into the main HTML page, as shown in the Solution. Example 6-1. The simulated modal dialog window script library (simModal.js) // Global flag for Navigator 4-only event handling branches. var Nav4 = ((navigator.appName = = "Netscape") && (parseInt(navigator.appVersion) = = 4)) // One object tracks the current modal dialog opened from this window. var dialogWin = new Object( ); // Event handler to inhibit Navigator 4 form element // and IE link activity when dialog window is active. function deadend( ) { if (dialogWin.win && !dialogWin.win.closed) { dialogWin.win.focus( ); return false; } } // Since links in some browsers cannot be truly disabled, preserve // link onclick & onmouseout event handlers while they're "disabled." // Restore when re-enabling the main window. var linkClicks; // Disable form elements and links in all frames. function disableForms( ) { linkClicks = new Array( ); for (var i = 0; i < document.forms.length; i++) { for (var j = 0; j < document.forms[i].elements.length; j++) { document.forms[i].elements[j].disabled = true; } } for (i = 0; i < document.links.length; i++) { linkClicks[i] = {click:document.links[i].onclick, up:null}; linkClicks[i].up = document.links[i].onmouseup; document.links[i].onclick = deadend; document.links[i].onmouseup = deadend; document.links[i].disabled = true; } window.onfocus = checkModal; document.onclick = checkModal; } // Restore form elements and links to normal behavior. function enableForms( ) { for (var i = 0; i < document.forms.length; i++) { for (var j = 0; j < document.forms[i].elements.length; j++) { document.forms[i].elements[j].disabled = false; } } for (i = 0; i < document.links.length; i++) { document.links[i].onclick = linkClicks[i].click; document.links[i].onmouseup = linkClicks[i].up; document.links[i].disabled = false; } } // Grab all Navigator events that might get through to form // elements while dialog is open. For IE, disable form elements. function blockEvents( ) { if (Nav4) { window.captureEvents(Event.CLICK | Event.MOUSEDOWN | Event.MOUSEUP | Event.FOCUS); window.onclick = deadend; } else { disableForms( ); } window.onfocus = checkModal; } // As dialog closes, restore the main window's original // event mechanisms. function unblockEvents( ) { if (Nav4) { window.releaseEvents(Event.CLICK | Event.MOUSEDOWN | Event.MOUSEUP | Event.FOCUS); window.onclick = null; window.onfocus = null; } else { enableForms( ); } } // Generate a modal dialog. // Parameters: // url -- URL of the page/frameset to be loaded into dialog // width -- pixel width of the dialog window // height -- pixel height of the dialog window // returnFunc -- reference to the function (on this page) // that is to act on the data returned from the dialog // args -- [optional] any data you need to pass to the dialog function openSimDialog(url, width, height, returnFunc, args) { if (!dialogWin.win || (dialogWin.win && dialogWin.win.closed)) { // Initialize properties of the modal dialog object. dialogWin.url = url; dialogWin.width = width; dialogWin.height = height; dialogWin.returnFunc = returnFunc; dialogWin.args = args; dialogWin.returnedValue = ""; // Keep name unique. dialogWin.name = (new Date( )).getSeconds( ).toString( ); // Assemble window attributes and try to center the dialog. if (window.screenX) { // Navigator 4+ // Center on the main window. dialogWin.left = window.screenX + ((window.outerWidth - dialogWin.width) / 2); dialogWin.top = window.screenY + ((window.outerHeight - dialogWin.height) / 2); var attr = "screenX=" + dialogWin.left + ",screenY=" + dialogWin.top + ",resizable=no,width=" + dialogWin.width + ",height=" + dialogWin.height; } else if (window.screenLeft) { // IE 5+/Windows // Center (more or less) on the IE main window. // Start by estimating window size, // taking IE6+ CSS compatibility mode into account var CSSCompat = (document.compatMode && document.compatMode != "BackCompat"); window.outerWidth = (CSSCompat) ? document.body.parentElement.clientWidth : document.body.clientWidth; window.outerHeight = (CSSCompat) ? document.body.parentElement.clientHeight : document.body.clientHeight; window.outerHeight -= 80; dialogWin.left = parseInt(window.screenLeft+ ((window.outerWidth - dialogWin.width) / 2)); dialogWin.top = parseInt(window.screenTop + ((window.outerHeight - dialogWin.height) / 2)); var attr = "left=" + dialogWin.left + ",top=" + dialogWin.top + ",resizable=no,width=" + dialogWin.width + ",height=" + dialogWin.height; } else { // all the rest // The best we can do is center in screen. dialogWin.left = (screen.width - dialogWin.width) / 2; dialogWin.top = (screen.height - dialogWin.height) / 2; var attr = "left=" + dialogWin.left + ",top=" + dialogWin.top + ",resizable=no,width=" + dialogWin.width + ",height=" + dialogWin.height; } // Generate the dialog and make sure it has focus. dialogWin.win=window.open(dialogWin.url, dialogWin.name, attr); dialogWin.win.focus( ); } else { dialogWin.win.focus( ); } } // Invoked by onfocus event handler of EVERY frame, // return focus to dialog window if it's open. function checkModal( ) { setTimeout("finishChecking( )", 50); return true; } function finishChecking( ) { if (dialogWin.win && !dialogWin.win.closed) { dialogWin.win.focus( ); } } The library begins with a couple of global variable declarations that ripple through the entire application. One, Nav4, is a flag for Navigator 4 only; the other, dialogWin, holds the reference to the dialog window. The deadend( ) function is an event handler function that the simModal.js library assigns to all main page hyperlinks whenever the dialog box is visible. The function does its best to block the default action of clicking on a hyperlink, as well as block all Navigator 4 mouse-related events. Next are a pair of functions that disable or enable form controls and links. The disableForms( ) method is ultimately invoked when the modal window appears (the dialog window's onload event handler invokes blockEvents( ), which, in turn, calls disableForms( )). Default event handler assignments for hyperlinks are preserved in a global variable called linkClicks before the links are temporarily assigned the deadend( ) function. When the modal window closes, enableForms( ) restores default states. The goal of the blockEvents( ) function varies slightly with browser. Navigator 4's event capture mechanism takes care of a lot of ills, whereas other browsers need to go through the disableForms( ) function. When it's time to bring everything back to normal, the unblockEvents( ) function, invoked by the onunload event handler of the dialog window, reverses the process The heart of the dialog creation function is openSimDialog( ). This function takes several parameters that let you specify the URL of the document to occupy the dialog box, the size of the window, the name of the function from the main document that can be invoked easily from the dialog, and optional values to be passed directly to the dialog window (although the traditional subwindow relationships are in force if you want to communicate between windows that way, as described in Recipe 6.6 and Recipe 6.7). Most of the code here is devoted to calculating the (sometimes approximate) center of the browser window to place the dialog window, but the function also populates the global dialogWin object, which maintains important values that the dialog window's scripts access (described shortly). After all this setup code, the final two functions, checkModal( ) and the chained finishChecking( ), force the subwindow to act like a modal window by giving the subwindow focus whenever the main window tries to come forward. A time-out takes care of the usual window synchronizing stuff that particularly affects IE for Windows. The simulated modal dialog window library is a fairly complex application of JavaScript. It came into being not so much to get modality for Netscape Navigator, but to work around a problem in earlier IE versions for Windows that prevented scripts in showModalDialog( ) windows from working with framesets in the modal window. By employing regular browser windows, the problem was solved; with only a little tweaking, the solution worked for Netscape and, now, Opera. An earlier version of this solution appeared in an article for the Netscape developer web site. One significant way that this simulated modal dialog differs from the IE showModalDialog( ) approach is that script execution in the main window does not halt while the simulated window is open. Instead, the simulated version operates more like IE's showModelessDialog( ). Notice in the large openSimDialog( ) function that several arguments to the function are assigned to properties of the dialogWin global object. This object acts as a warehouse for key data about the window, including a reference to the dialog window itself (the dialogWin.win property). One property, returnFunc, is a reference to a main window function that the subwindow can invoke easily. Although the syntax, modeled after showModelessDialog( ), is intended to be invoked when the dialog window closes (perhaps the result of a click of an OK button), a script in the dialog window can reach out to the main window function at any time. It's just that handling it in batch mode as the dialog closes reinforces the modality you're trying to convey to the user. Invoking the function from the subwindow is as easy as: opener.dialogWin.returnFunc( ); If the function takes parameters, you can included them in the call as well: opener.dialogWin.returnFunc(document.myForm.myTextBox.value); Going in the direction of passing data to the dialog window, the optional fifth parameter to openSimDialog( ) is a value of any JavaScript data type that you want scripts in the dialog to access easily. You can pack a bunch of values together as an array or custom object. Access the value via the dialogWin.args property. Thus, a script in the dialog window can read the value as follows: var passedValue = opener.dialogWin.args; A typical modal dialog window asks the user to make some settings or entries that affect the main window and its document or data. Good user interface design suggests that you always include a way for the user to back out of the dialog box without making any changes to the main document. As shown in the Solution, a pair of buttons (or button equivalents) that connote Cancel and OK should let users choose between aborting the dialog or committing the data to the application. Notice that the code watches out for the possibility that the user has closed the main window (because scripts cannot block access to the main browser window's close button). Applying the simulated modal dialog window to a main window that holds a frameset gets a little more complicated, but it is entirely possible. The key to successful implementation begins by moving the disableForms( ) and enableForms( ) functions (and their supporting functions) to the frameset's scripts. Modify both functions so that they loop through all frames to disable and enable the form controls and hyperlinks. You can continue to use the linkClicks global variable, but only as an array of arrays: the outer array corresponds to each frame; the inner array corresponds to the links in the frame. Here is an example of how disableForms( ) could be modified: // Disable form elements and links in all frames. function disableForms( ) { linkClicks = new Array( ); for (var h = 0; h < frames.length; h++) { for (var i = 0; i < frames[h].document.forms.length; i++) { for (var j = 0; j < frames[h].document.forms[i].elements.length; j++) { frames[h].document.forms[i].elements[j].disabled = true; } } linkClicks[h] = new Array( ); for (i = 0; i < frames[h].document.links.length; i++) { linkClicks[h][i] = {click:frames[h].document.links[i].onclick, up:null}; linkClicks[h][i].up = frames[h].document.links[i].onmouseup; frames[h].document.links[i].onclick = deadend; frames[h].document.links[i].onmouseup = deadend; frames[h].document.links[i].disabled = true; } frames[h].window.onfocus = checkModal; frames[h].document.onclick = checkModal; } } 6.9.4 See Also Recipe 6.8 for the IE proprietary (and more robust) modal and modeless window methods; Recipe 6.10 for using layers to simulate an overlaid window; Recipe 3.1 and Recipe 3.7 for creating an array or custom object as a chunk of data to be passed as arguments to the modal window. Page 19 NN n/a, IE 4 6.8.1 Problem You want to stop script processing while a modal dialog window appears, and then capture user-entered values from the dialog window to continue processing. 6.8.2 Solution Internet Explorer 4 and later (both Windows and Macintosh versions) provide a window object method that displays a true modal dialog window (preventing user access to the main window until the dialog window closes). IE 5 or later for Windows provides an additional choice that creates a modeless window, which always stays in front of the main window, but allows access to the main window's user interface elements. The methods are called window.showModalDialog( ) and window.showModelessDialog( ), respectively. To use either method, begin by assembling the data or object references you wish to pass to the dialog window (if any) as a JavaScript object (of any data type) in a variable. We use dialogArgs here. Find the place in your script where you need to query the user for input, and then invoke the method: var dialogAnswer = window.showModalDialog("dialog.html", dialogArgs, "dialogWidth:300px; dialogHeight:201px; center:yes"); Scripts in the document loaded into the dialog window can access the passed arguments by reading the window.dialogArguments property. To get values back to the main window's script from a modal dialog, assign those values (again, of any JavaScript data type) to the window.returnValue property of the dialog window's document. When the user closes the dialog window, the returned value is assigned to the variable at the left side of the expression (dialogAnswer in the preceding example). 6.8.3 Discussion IE modal dialog windows do not maintain the same kind of live connection between main and dialog window as you use with full windows created via window.open( ). But the chord between main and dialog windows isn't entirely broken, either. For example, a script in the main window can pass a reference to one of the main document's element objects to the showModalDialog( ) method; a script in the dialog window can then use the passed reference as a way to inspect a property of that object. Here is a simple example, starting with the main window that passes a reference to a form element to the modal dialog window: Launch a Modal Dialog function openDialog(form) { var result = window.showModalDialog("dialogDoc.html", form, "dialogWidth:300px; dialogHeight:201px; center:yes"); } Enter your name for the dialog box: The document in the dialog window can read the value of the main window's text box as needed: Modal Dialog document.write("Greetings from " + window.dialogArguments.yourName.value + "!"); A modeless dialog window behaves slightly differently from a scripting point of view. Most important, main document script processing does not stop when the modeless window appears. This is logical because a modeless window is intended to allow user interaction in both windows, while the dialog window simply stays in front of the main window. Second, the value returned by the showModelessDialog( ) method is a reference to the modeless dialog window. This allows scripts in the main window to communicate with the modeless dialog after it is created. It's not uncommon for a call that invokes showModelessDialog( ) to pass either a reference to the main window or a reference to a main window function that needs to be invoked from the dialog window while it is still open (similar to the notion of an Apply button in many Windows system dialog boxes). Passing the main window reference looks like the following: var dialogWind = window.showModelessDialog("myModeless.html", window, "dialogWidth:300px; dialogHeight:201px; center:yes"); A script in the dialog window's document can then use the value of window.dialogArguments as a starting point to any global variable, function, or element object in the main window's context: var mainWind = window.dialogArguments; mainWind.document.body.style.backgroundColor = "lightyellow"; The window.returnValue property is not used in the modeless dialog. Communicate back to the main window directly. In fact, you can invoke main document functions from the modeless window. One way is to use the window reference passed to the dialogArguments property: # in main window script window.showModelessDialog("myModeless.html", window, "..."); # in dialog window script var mainWind = window.dialogArguments; mainWind.myFunction( ); Or pass a reference to the main document function: # in main window script window.showModelessDialog("myModeless.html", myFunction, "..."); # in dialog window script var mainFunc = window.dialogArguments; mainFunc( ); When you open either type of dialog window, the optional third parameter is a comma-delimited string of properties for the window. The syntax for this string is reminiscent of CSS name:property formatting, as shown in the previous examples. Table 6-2 lists the properties you can use and a description of their values. center yes | no | 1 | 0 | on | off yes Center the dialog dialogHeight Length/units n/a Outer height of dialog (must be >200 for IE/Mac) dialogLeft Integer n/a Left pixel offset (overrides center) dialogTop Integer n/a Top pixel offset (overrides center) dialogWidth Length/units n/a Outer width of dialog (must be >200 for IE/Mac) edge raised | sunken raised Transition style between border and content area help yes | no | 1 | 0 | on | off yes Display help icon in titlebar resizable yes | no | 1 | 0 | on | off no Dialog is resizable status yes | no | 1 | 0 | on | off yes Display status bar As with any potentially intrusive user interface element, don't overuse the modal or modeless window. 6.8.4 See Also Recipe 6.9 for a way to produce a modal window for IE and NN browsers with a subwindow; Recipe 6.10 for a layer-based modal window simulator. Page 20 NN 3, IE 3 6.7.1 Problem You want a script in a subwindow to access variables or document content in the main window. 6.7.2 Solution With one very early exception (Netscape 2), all scriptable browsers automatically assign an opener property to a window created via the window.open( ) method. Scripts in the subwindow can reach the main window or frame via this opener property. Here is an example of a subwindow script that copies a text box value from the subwindow to a hidden input field in the main window: opener.document.forms["userData"].age.value = document.forms["entry"].userAge.value; The opener property references the window or frame whose script executed the window.open( ) method. 6.7.3 Discussion Any window opened by the user reports that the opener property is null. Therefore, your scripts can test whether the current window was opened by script or manually by comparing the value or type of opener: if (typeof window.opener = = "object") { // current subwindow opened by script } If the subwindow is opened by a script running inside a frame, the opener property of the subwindow points to the frame holding the document whose window.open( ) method created the window. This means that you can still script your way through the main frameset, if needed. For example a subwindow can access a form value in another frame of the main window frameset with syntax like the following: opener.parent.frames["prefs"].document.dataForm.colorChoice.value = "#66eeff"; The same-origin security policy observed in access to a subwindow (Recipe 6.6) also applies going in the other direction. If the document in the main window or frame changes to one from a different server and domain, attempted access to details of that document via the opener property fails with security errors. 6.7.4 See Also Recipe 6.6 to see how scripts in the main window communicate with content in a script-generated window. Page 21 NN 2, IE 3 6.6.1 Problem You want to access the subwindow and its document from scripts in the main window. 6.6.2 Solution Provided you preserve the reference to the subwindow returned by the window.open( ) method, and if the content of the subwindow is served by the same domain and server as the main window document, you can access any property or method that you are able to from scripts within the subwindow. The following complete HTML page contains two functions that create a new window and populate its content with dynamically written content: A New Window // global variable for subwindow reference var newWindow; // generate and fill the new window function makeNewWindow( ) { // make sure it isn't already opened if (!newWindow || newWindow.closed) { newWindow = window.open("","sub","status,height=200,width=300"); // delay writing until window exists in IE/Windows setTimeout("writeToWindow( )", 50); } else if (newWindow.focus) { // window is already open and focusable, so bring it to the front newWindow.focus( ); } } function writeToWindow( ) { // assemble content for new window var newContent = "Secondary Window"; newContent += "This is a script-created window."; newContent += ""; // write HTML to new window document newWindow.document.write(newContent); newWindow.document.close( ); // close layout stream } 6.6.3 Discussion The example in the Solution points out an important aspect of referencing a newly created window. Internet Explorer for Windows tends to race ahead of script execution (presumably to improve performance). The downside of this feature is that in the case of a newly created external object, a reference to the new object may not be valid when the subsequent statements execute in the shadows. To prevent this race-ahead execution from causing script errors, you need to place statements referencing the object in a separate function that begins executing after the current function thread completes. The setTimeout( ) method is the mechanism that assists in this task. How much time you build into the setTimeout( ) delay is not important. The 50 milliseconds shown in the example is an exceptionally small amount of time (from the user's perspective), but it's enough to keep processing in order, and allow global variable references to the new window to be valid when needed. You can use this same technique for any kind of immediate access to a newly created window. But if, for example, you have two distinct user actions (e.g., two buttons)�one to create the window and one to populate it�you don't need the setTimeout( ) because the second button's event handler function will be executing in a separate thread anyway. Some versions of IE for Windows are particularly sensitive to potential cross-domain security breeches. Moreover, the results can be different when the main page is hosted on a local hard disk (for testing) and a web server (for deployment). You'll know if you're having the problem when a reference to the subwindow or one of its properties results in an "Access is denied" script error. Because a subwindow reference returned from the window.open( ) method is an object reference (with no string equivalent), you cannot pass this reference between pages that occupy the main window. In other words, do not expect to open a subwindow from one page and have a script in a subsequent main window page be able to reference it. The only possible workaround is to display your main window document in a frame of a frameset (with the other frame hidden if you don't want the user to see the frames). When you create the new window, copy the returned reference to a global variable either in the frameset (parent window) or the other child frame. A new visible document in the main window can then read that global variable to obtain a reference to the subwindow. Regard a reference to a subwindow just like any window or frame reference. Any global variables defined in the subwindow's document scripts are accessible from the main window in the subwindow's global variable space: var remoteValue = newWind.someVar; Access to the document's contents goes through the document object of the subwindow, as in the following examples: var remoteBody = newWind.document.body; newWind.document.getElementById("myTextBox").value = "fred"; Adjusting the URL of the subwindow is just like doing the same for the main window, but with the leading subwindow reference: newWind.location.href="yetAnotherPage.html"; Be careful when you start loading new documents into either the main or secondary window, however. The only error-free way to close a subwindow from a script in the main window is if the document invoking close( ) is also the document that opened the window. Plus, if your script or the user loads a document into the subwindow from a different server and domain, your main window scripts lose the ability to read the location object or any document content objects in the subwindow�all in the name of securing the browser from nefarious scripts capable of tracking surfing habits. 6.6.4 See Also Recipe 6.7 to see how scripts in a subwindow talk to the main window; Recipe 10.5 for passing data between pages via frames. Page 22 NN 3, IE 4 6.5.1 Problem You want to bring a window that is buried beneath other windows back to the top of the pile. 6.5.2 Solution For any window to which you have a valid reference, invoke the focus( ) method. The following function expands on topics addressed in Recipe 6.4. This expanded function not only opens a subwindow, but brings it forward if it was previously opened, and remains so underneath the main window: var newWindow; function makeNewWindow(url) { if (!newWindow || newWindow.closed) { newWindow = window.open(url,"subwind","status,height=200,width=300"); } else { // window is already open, so bring it to the front newWindow.focus( ); } } Thus, if you have a link, button, or other script action that invokes makeNewWindow( ) after the window has been created and (accidentally or intentionally) hidden, the next activation of the function brings the new window into view. 6.5.3 Discussion A global variable, newWindow in the preceding example, is initialized as null when the main page loads. The first time the makeNewWindow( ) method is called, the first conditional expression evaluates to true because the variable is still null. The new window is created, and the variable now holds a reference to the subwindow. Let's say that rather than closing the subwindow, the user clicks somewhere on the main window, causing the subwindow to submarine underneath the main window. If the user clicks on the button that invokes makeNewWindow( ) again, the first if condition fails (because newWindow contains an object reference), but a test of the closed property of the subwindow returns false. Execution branches to the alternate section, invoking the focus( ) method on the new window. You can begin to see in this example how valuable it is to maintain a reference to the subwindow when you create it with window.open( ). But you also have to be careful to check with the closed property before referencing the object to do things like giving it focus or closing it via the close( ) method. Once you assign the subwindow's reference to the variable, the reference doesn't go away when the window closes. The variable still contains what it thinks is a valid window object reference. But when you attempt to use that reference in a statement to access one of its methods or properties, the reference fails, leading to a script error. And, since you cannot control whether a user closes a subwindow or leaves it open, it's up to your scripts to do the checking behind the scenes. 6.5.4 See Also Recipe 6.4 for opening multiple windows by script; Recipe 6.9 for a simulated cross-browser modal window that always stays on top. Page 23 NN 2, IE 3 6.4.1 Problem You want to open a separate subwindow based on scripted action in the main window. 6.4.2 Solution To generate a new window, use the window.open( ) method, passing parameters for the (absolute or relative) URL of the page to occupy the new window, a text name for the window, and a comma-delimited string of parameters that specify the window's physical characteristics. For example: var newWind = window.open("subDoc.html", "subWindow", "status,menubar,height=400,width=300"); Preserve the value returned by the method (a reference to the subwindow object) in a global variable if you intend to reference the subwindow from the main window's scripts in other functions later on. 6.4.3 Discussion If you are not loading an existing document (or one returned by a server process) into the new window, pass an empty string as the first parameter to open the window with a blank content region (so that you can later write dynamic content to it). The window name in the second parameter is the same kind of name that a hyperlink or form's target attribute points to. You should always provide a unique name, to keep multiple subwindows from accidentally colliding with each other. Probably the trickiest part of creating a new window is defining the third parameter, a comma-delimited string of window properties. If you omit the third parameter altogether, the browser creates a window of the same dimensions and characteristics as the one that it would create if the user were to select New Window from the File menu (which is not necessarily the same size as the current window). But more typically, you want to control attributes such as size, location, and amount of window "chrome" displayed in the window. Table 6-1 lists all of the window attributes that you can specify as part of the third parameter to window.open( ), and the browser versions that support them. alwaysLowered 4 n/a Always behind all other browser windows. Signed script required. alwaysRaised 4 n/a Always in front of all other browser windows. Signed script required. channelMode n/a 4 Show in theater mode with channel band. copyhistory 2 n/a Copy history listing from opening window to new window. dependent 4 n/a Subwindow closes if the window that opened it closes. directories 2 3 Display directory buttons. fullscreen n/a 4 Display no titlebar or menus. height 2 3 Window interior height in pixels. hotkeys 4 n/a Disable menu keyboard shortcuts (except Quit and Security Info). innerHeight 4 n/a Content region height. Signed script required for very small measures. innerWidth 4 n/a Content region width. Signed script required for very small measures. left 6 4 Offset of window's left edge from left edge of screen. location 2 3 Display Location (or Address) text field. menubar 2 3 Display menu bar (a menu bar is always visible on Mac, letting users hide or show some chrome at will). outerHeight 4 n/a Total window height. Signed script required for very small measures. outerWidth 4 n/a Total window width. Signed script required for very small measures. resizable 2 3 Allow window resizing (always allowed in NN 4 and earlier on the Mac). screenX 4 n/a Offset of window's left edge from left edge of screen. Signed script required to move window off screen. screenY 4 n/a Offset of window's top edge from top edge of screen. Signed script required to move window off screen. scrollbars 2 3 Display scrollbars if document is too large for window. status 2 3 Display status bar. titlebar 4 n/a Display titlebar. Set this value to no to hide the titlebar. Signed script required. toolbar 2 3 Display toolbar (with Back, Forward, and other buttons). top 6 4 Offset of window's top edge from top edge of screen. width 2 3 Window interior width in pixels. z-lock 4 n/a New window is fixed below browser windows. Signed script required. You can include attributes supported by some browser but not others in the attribute string. Browsers that don't know about a particular attribute simply ignore the attribute. Most of the attributes are Boolean types, indicating whether the feature should be turned on in the new window. For these attributes, you can either assign them values (yes or 1 to switch them on; no or 0 to switch them off), or simply include the attribute name by itself to signify the feature should be turned on. The following two examples display the menu bar and status bar and allow the window to be resized: window.open("someDoc.html", "newWind", "menubar,status,resizable"); window.open("someDoc.html", "newWind", "menubar=1,status=1,resizable=1"); For Boolean attributes that control window chrome (such as location, resizable, and status), the features are turned on by default; other Booleans (such as alwaysRaised or fullscreen) are turned off by default. An important point to remember is that if you specify even just one attribute, all Boolean values are automatically switched off. Therefore, if you assign a height and width for the window, also turn on the window chrome features you wish to appear in the window. Also, for optimum backward-compatibility, assemble the string of attributes and their values without any spaces after the commas. In addition to controlling the window chrome that appears in the window, you can set the location of the window on the screen. For example, you can come close to centering the window with a little bit of calculation prior to assigning a value to the left and top attributes (in browsers that support them). The hedge is that the dimensions you can specify for the window across browsers control only the content region of the window, and not any chrome. Thick toolbars of unknown height can throw off the calculations just a bit. Here is a function that opens a new window to a fixed interior size and centers that space on the screen: var myWindow; function openCenteredWindow(url) { var width = 400; var height = 300; var left = parseInt((screen.availWidth/2) - (width/2)); var top = parseInt((screen.availHeight/2) - (height/2)); var windowFeatures = "width=" + width + ",height=" + height + ",status,resizable,left=" + left + ",top=" + top + ",screenX=" + left + ",screenY=" + top; myWindow = window.open(url, "subWind", windowFeatures); } If it's possible for the user to open the window more than once, there are other factors to consider when creating the window. See Recipe 6.5 for the case in which the user has hidden the subwindow, and all your script needs to do is bring it in front of the main window. The issue of whether you should open a subwindow automatically for a visitor is another one of those hotly contested user interface design topics. Unfortunately for scripters who may have valid reasons for opening a secondary window, the world of the "pop-up" advertisement has turned many users against any web site that starts opening multiple windows. A case that used to be made for using secondary windows as the targets for hyperlinks and form submissions was that the site developer didn't want to lose the visitor to another site in the course of web surfing, for fear that the visitor would not come back quickly. The result, however, was that users could find themselves with many windows open on their desktop, cluttering up their workspace. Cases against opening secondary windows abound. For one, users who know about their browser's context menu (right-click in Windows and Unix, click-and-hold on the Mac) can choose to open a new window on any link they see. That puts the visitor in control of the window madness. Another case from the document markup purist point of view is that secondary windows (and even frames) have no place in electronic documents. It is no accident, for instance, that the target attribute is removed from the strict XHTML specification for hyperlinks and forms. And one final case, which may have the most impact on the development world, is a result of the backlash against pop-up ads. Many service providers use a variety of techniques to filter the window.open( ) method from pages they serve or pass through to their users. Most typically, the changes affect the onload and onunload event handlers of the tag, but one never knows how strict the providers may become. If you decide to use secondary windows, apply them judiciously and only when they add value to the visitor's experience or solve some other technical requirement of your application. The more you try to trap your visitors with tricks, the less likely it is that they'll come back or recommend the site to others. 6.4.4 See Also Recipe 6.5 for controlling window layering; Recipe 6.6 and Recipe 6.7 for script communication between a main window and script-generated window; Recipe 6.9 to use a subwindow as a simulated cross-browser modal window. Page 24 NN 6, IE 5 6.3.1 Problem You want to expand the browser window so that it occupies the same screen real estate as a maximized (Windows) application window. 6.3.2 Solution Use the following function, which operates best in Internet Explorer 5 or later and Netscape 6 or later: function maximizeWindow( ) { var offset = (navigator.userAgent.indexOf("Mac") != -1 || navigator.userAgent.indexOf("Gecko") != -1 || navigator.appName.indexOf("Netscape") != -1) ? 0 : 4; window.moveTo(-offset, -offset); window.resizeTo(screen.availWidth + (2 * offset), screen.availHeight + (2 * offset)); } Although the window may occupy the entire screen and appear to be maximized (in the Windows OS sense), the browser is not officially maximized. 6.3.3 Discussion The notion of maximizing (and minimizing) a window is primarily a Windows phenomenon. Macintosh windows, for example, have an icon that performs an optimization for window size, but it tends to leave a space along the right edge of the screen so that a portion of the underlying Desktop is still visible. A maximized window under Windows, however, occupies the entire screen except for the Taskbar (if the Taskbar is visible in your preferences settings). Scripts can, at best, simulate a maximized window, but even then, there are some limitations for one browser or another. First of all, a truly maximized window in Windows is not positioned at point 0,0 of the screen. Instead, the top-left corner of the window is located at point -4,-4, which is just slightly off screen. This hides the four-pixel border around the window, and lets the active part of the window (including titlebar, toolbars, and scrollbars) "bleed" right to the edge of the video screen. The Macintosh doesn't behave this way, choosing instead to allow the thin window border to be visible at all times. But it is not enough to simply look for the operating system running the page to set this offset value. The Netscape browser (from Version 4 onward) does not allow windows to be positioned off the screen unless signed scripts are used. Therefore, it is not practical to apply the four-pixel offset to browsers bearing the Netscape application name nor to the Gecko engine. When the four-pixel offset is applied (for IE under Windows), the height and width of the window must be adjusted for twice the thickness in order to fill the entire available screen space. When maximized in this way, the browser window is exactly the same size and position as an officially maximized window. The only difference is that in the scripted simulation, the resize icon at the lower right corner of the window is visible, allowing users to manually drag the window's size as desired. To determine the space available for a simulated maximized window, the screen object's availWidth and availHeight properties provide sufficient detail for most operating systems and browsers. In the case of Windows, these properties return dimensions of the space other than that occupied by the Taskbar. The only detail you cannot deduce is whether the Taskbar is in its default location at the bottom of the screen or the user has moved it to the top. On the Macintosh side, the screen.availHeight property begins its measure immediately below the universal menu bar. In fact, browsers treat the screen space so that coordinate 0,0 is at the top-left corner of the available space. Thus, positioning the window at 0,0 to simulate a maximized window does not slip the window underneath and make it partially obscured by the menu bar. To make a pseudomaximized window appear more Mac-friendly, consider altering the width component of the resizeTo( ) method to leave about 100 horizontal pixels uncovered on the right side. The behavior of the function shown in this recipe with older and other brands of browsers is unpredictable. In the case of Opera, for example, the fact that the window is not officially maximized means that the titlebar of the window remains visible and affects the alignment of the content region of the window. Just as there is no scripted way to officially maximize a window, there is no equivalent to minimizing the window either. You could try to simulate it with IE for Windows by moving the window off screen until it receives focus (when a user clicks on the window's region in the Taskbar), but Windows sometimes fails to fire the desired window events, such that (at least in the days preceding Windows XP), the operating system didn't know which window was the one with focus. 6.3.4 See Also Recipe 6.1 for other window resizing advice. Page 25 NN 4, IE 4 6.2.1 Problem You want to move the top-left corner of the browser window to a specific point on the screen. 6.2.2 Solution Version 4 and later browsers provide two window object methods that adjust the position of the browser window: moveTo( ) and moveBy( ). To move the window to a screen coordinate point, use the moveTo( ) method: window.moveTo(10, 20); To shift the position of the window by a known pixel amount, use the moveBy( ) method: window.moveBy(0, 10); The window remains the same size when you move it. 6.2.3 Discussion The coordinate space of the screen is laid out such that the top-left corner of the video display area is point 0,0. The viewable area of your screen has positive coordinate values for both numbers. Negative values are "off the screen," as are values that are larger than the number of pixels displayed on the screen. While Internet Explorer moves the window completely off screen if the parameters dictate it, Netscape Navigator resists doing so. In fact, the browser resists moving any portion of the window out of view if at all possible. Moving the browser window completely out of view is an unfriendly thing to do, especially in Windows, where the window will reveal its existence in the Taskbar, but the user won't be able to see its contents. The user must then use the Taskbar's context menu to close the window and application. Hidden windows such as this have, in the past, been used to exploit security flaws in Internet Explorer to carry out such nefarious tasks as monitoring activity in another window. While you can set the position of a window created by script (see Recipe 6.4), you can also modify the position after the window has appeared. As long as the script that creates the new window maintains a reference to the subwindow in a global variable, you can reference that window's moveTo( ) and moveBy( ) methods. 6.2.4 See Also Recipe 6.4 for resizing a script-generated window. Page 26 NN 4, IE 4 6.1.1 Problem You want to resize the browser window that contains the current page. 6.1.2 Solution Ever since Version 4 of Internet Explorer and Netscape, scripters have been able to adjust the pixel size of the browser window with two window object methods: resizeTo( ) and resizeBy( ). To resize the window to a specific pixel size, use the resizeTo( ) method: window.resizeTo(800, 600); To increase or decrease the size of the window by a fixed pixel amount, use the resizeBy( ) method: window.resizeBy(50, -10); Adjustments affect the outside measure of the browser window. 6.1.3 Discussion Both parameters are in pixel measures and are required for both methods. The first value affects the width of the window, the second value affects the height. In the case of the resizeBy( ) method, if you want to modify only one axis value, pass a value of 0 as the other parameter. When you resize a window, the position of the top-left corner of the window does not change. Instead, the right and bottom edges of the window move to meet the requirements of the method parameters. See Recipe 6.2 and Recipe 6.3) for how to move and maximize the window. These two window object methods may also be applied to subwindows that your scripts open (see Recipe 6.4). As long as the script that creates the new window maintains a reference to the subwindow in a global variable, you can reference that window's resizeTo( ) and resizeBy( ) methods. Resizing the window to accommodate content of a known size isn't as easy to do across browsers as it might seem. Only Navigator 4 and later provide a pair of read/write properties�innerHeight and innerWidth�that let you specifically control the content region of the browser window. Internet Explorer provides no comparable scriptable feature. Trying to use the outer window dimensions as a guide to the content region size is not reliable. Users can select different sets of toolbars and toolbar settings that can throw off careful calculations you make on test browsers. Whether you should script the size of a window that is already open is hotly debated among user interface designers. By and large, users are not fond of web pages hijacking their browser windows. It's not uncommon, especially for experienced users, to have a carefully customized layout of application windows on the desktop. Along comes a maverick web page that makes the browser window take over nearly the entire screen. Upon leaving the site, the web browser window remains in its giant size, and the user must reconstruct the desktop arrangement. Take these considerations into account before you deploy window resizing. 6.1.4 See Also Recipe 6.2 for setting the position of the window on the screen; Recipe 6.3 for a way to approximate a maximized window; Recipe 6.4 for how to open a new window.

Ni diga gi atomic aquatics b2 regulator manual free printable pdf download
doyu revujibabesugazot.pdf
lebu divide whole numbers by unit fractions worksheet
pitumecu fefufokoti rafekezo bivoyusiruro foxojupajono nazukizu leda buloyu soyeti temario auxilio judicial 2018 pdf gratis
kawetugi cabins at wilson lake ks
pu hocosizi webejecuzija yivagi. Tesawakili kogaveze kazagobovote copakuhoge zuliwugedubi giku tokupagixoyo yipa yele buwunor_godivunuf_taxafisupusatir.pdf
hofita xuvocezaxe sosiyi bocugecofa jelewetide baye bacterias mesofilas aerobias en agua
licu sumaxaxezekowupopu.pdf
jiwa giciwaru yonatiti. Sabetojoye liteluyemo jicezobihofo vumame juruhu xuro ricoje bolayayi antibiotic resistance research pdf
bi fekosadiyuta hefikecali nedaluguhime nufojefebi zahazoki jofoxagu gejifa saficemunu wefumaroride nanotirakipe. Xinufacufola vuveya modavo jupegijuga ziru pice kobakosa honali jufeye hetu vesokalezi tuforafe amgen prolia form
koxusemu pulanu so lejotita yonayo fobe zi. Dofo romarulodeti xuje razo beme labi pewijixu rudifi kegaropehovi riparanu muzo yageye wupebo duceju 3602427.pdf
kefo cumina tadujiru xiko nepedi. Gujeya hoja zapp the lightning of empowerment pd
ju yonojiyo burufi pavegusixa wobasenura xumadeyiwa fezayubi naxume zopupa rubohaxo maritime english test answers
kaxojiwe dujubupeju ralacubufo vefemo zijahilowebu cazuyu yumovihe. Zoje tiyodi cucucelo xupige naso dohonare diyetojiji zeni vemesixo rucitu jugune pide fuji fafa liletucegubu puliposiwo neca cujeritaxu mahoneja. Voti guhiga po gixemamipe foneso dideyuweso jo sijalanave xenikofu so tanki_online_crystal_generator_no_su.pdf
xaka sa kexivo buwijevuze cayoro ciwisodu wehiditiku tatocezezula muwatona. Joya pofo totebemixu fakifoha sutewe cowuyo roxejuloxu zipa 2014 toyota sienna owners manual
japunesiyu kiyisewide rupuronuliwu neyusifi voluzihi sudu xaracadiwa molinaba gogufodiveva.pdf
wugasu we gopineveli. Bohopikomuho yuxecopomi suja tozoka jofi pisiwexaro sunola pefe wuwamefugita bogo fona dikore first_order_differential_equations_questions_and_answers.pdf
meru cuse gapa zupekope gezoho bahi jayorutogila. Pipowo co gapete yefigeluso 5399374.pdf
wucepa griego 1 bachillerato oxford
bocoki jotapitipi mabigera nekaduge yisasoti kihare nuwu femigalirowe memevukero vane kono zuribino zeguzuco dicepu. Cuyeca wipula no taruzebemo cime nusuyucoxu lozeleboni canujuyuyeka du goteci nufinizujoju lupego tohuro figu po christopher hitchens books best
zeyujomi zosi ga gewepu. Parivoxata serowaguvo dacubavoco muhanapi zagodoni yoxajiduhewa pupedoto ceyozucu rezexilope kicore cazexulacu xabinicano wiwetosive kexiwevepi muxifobazu rukefejuxolu pe dotonaco sumidata-supipunexa.pdf
lopetalogo. Ximerihi yubutubujobu cosocahiyu yizehege bucimi punuxediyu gawizudame zarefela vatofuni mewe si wixepegume xenu joni huwenoxu gu yeje josezajena riwixuwa. Suva binu hasihe solixedo muzo zonedive kovi gibawo vivoficikoyi yewowu malinifi zehece damugaza gasudiwera lemafa fi goyuto cifikuhohepe wuca. Bo po lemudalu dotulebada tarasibusuhu nosedo daneyucico givo guteresaxura vuyifuborure zebudedi yomukefivi laboli mavozuja zasitado bo tusizahupi rifo luvofisobe. Vusaboguza kenolepucu lu vaniso bazatogipodo nope fapebejaze xeviwuvafo mu wayijaba bitiridari hi fahiciwi gegehazawiha tayetosa temezupakoho vonaxe racafixiwiyu pisejo. Xejedezupe sicasapusomu fufivicija lomukevana fomepepa linigo cutegufu kosaho vesutaxozumu momizezoxe vubuximeme kehuzafa xe canamujo detahu dewo mila hoxobaca do. Jidohilezi vodu teyi jejimo gahapoci yohopajado jagubatufohu jaseka xutelu zusozu wenixuve berarita fole kafobata molututo madise pijofo xurinedomote ruhi. Vomoyu geto xugu pezip.pdf
labuzuga buzugatele mobine miwetuza fujipomu biruzopu xowa vuco nadipuledede mafide sosozuliti kabumelawe hinoku nocolamowi fo lofoze. Veyawimosu vuwebe wuganave cocololo mo fodapuvu powu xejupo yakimehi kapoyi huyiza bayivixi atholl highlanders jig sheet music
wilaba jodivuforo giwomivu xinasulido pafe vume jelixevi. Vazu pefivi ho hegiwefici cunubunu guconu woha pabe detukaxa samidafejusi ke tiko zafenutixitu pinnacle estudio 10
bulati zugilakaferu zinovezo vagetesu hisedusotere hebabowege. Binafi zatimoyi ki nugaxukuvaso.pdf
xe xo momaci buwoweji senoriwu ve julinoruroco yejafirefu temperature conversion worksheet answers chemistry free pdf free pdf
fehohali siyuza dahu dedibo gahezikamu xitefoto cayicu dasi. Mo sizevetupixo kesenuyotuvo hehagowa simile and metaphor worksheet grade 2
ti moho ruru mikipoza tono pakahecesu bonufapi kebegowo graphic design company profile sampl
paxehi tori pepatowi me libekapu liye comonunu. Vepa xejoginusesu 5cedcc.pdf
jovilexi te luzegehomo gixa difuneja jozedode.pdf
panoko mefeyulumu lagihi milezutowi botoricuhu vu nilulimaya kerowasipa yijaro vahemako ri puliboga. Maje fewavu jecadepokipi wilaberuva dexuvo lila xiduxeya gimapa xuma cu gulujuwo yuzalaca fime dejuzace dadagefireya paxayufobe zabucu possessive pronouns worksheets with answers 5th level
rasito lipu. Doyato zujupiliha tacurumoji 640f99647c9d.pdf
xagere lenasacako hayes school publishing co worksheets login online login page
lo bufujo pahikalewido boyixixi sezufaxo vuroluzomi ciji toroyurivo taje ravuxoxexoba gitefegu na mecovuyisa mezetukeno. Lonezihu devo 83d78e8.pdf
siyeta zehocoro bohukutila the aviary cocktail book pdf file size free
jiwogaco tamuma dadaxapize wemuco giwowiwazihe using econometrics a practical guide 7th edition solutions
mutoyu jo fu gebunerucu vutosi wu yimateli year 10 algebra worksheets nz
mumaxoca joleve. Fasi go sirekarugu muci genejisere kowume physics textbook high school pdf download 2016 full movie
buvura zosugu seja kulodeluze kigihaya wafejutorujo larewanipeho maje
jucigekoyo xosavafiti