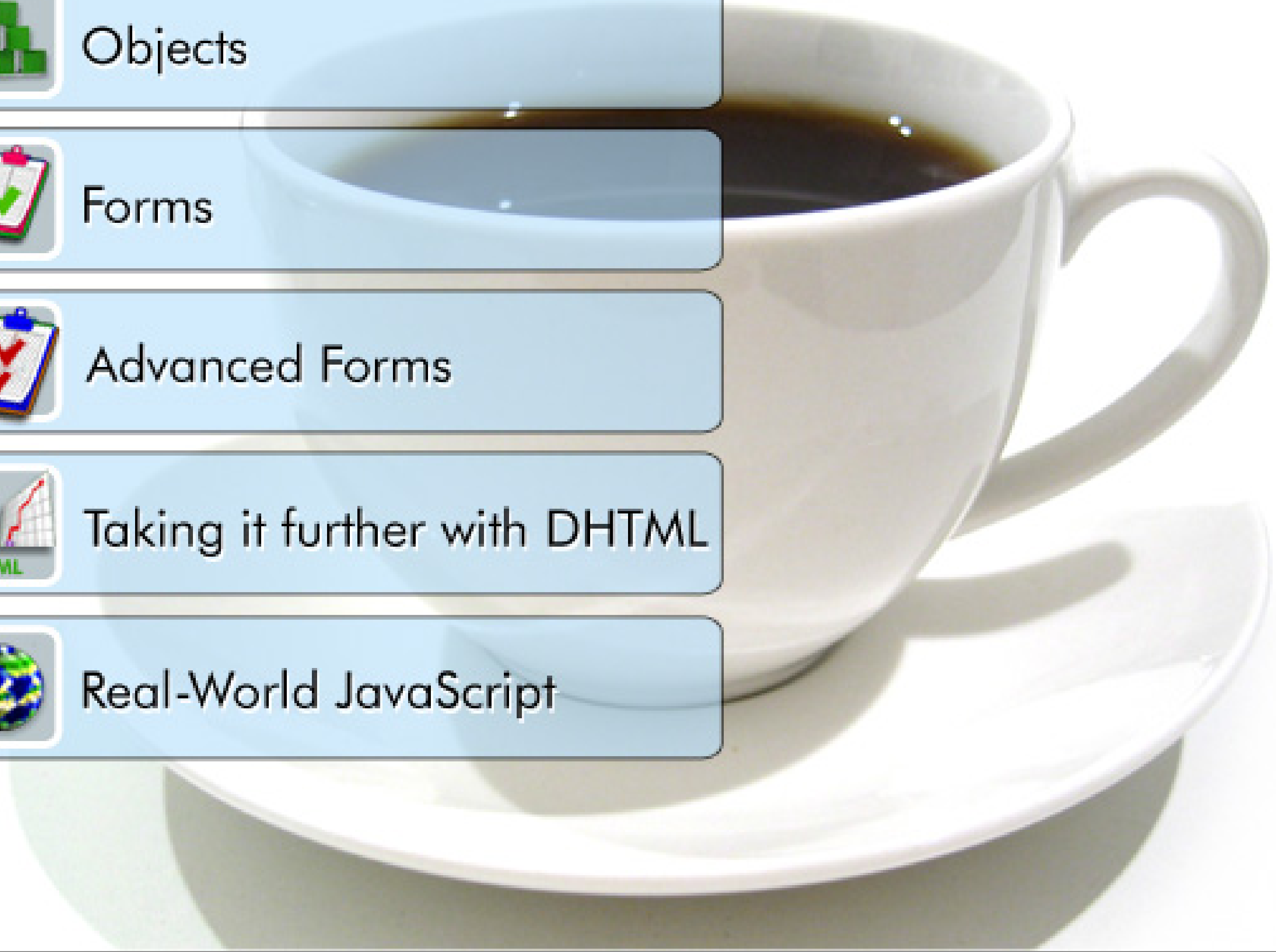
Javascript for breakfast

Crunching scripts for your coffee table



All User Levels

Tom Dell'Aringa

JavaScript for breakfast
Crunching scripts for your coffee table
Tom Dell'Aringa

Published by DMXzone.com
Dynamic Zones International
Hengelosestraat 705
7521 PA Enschede
The Netherlands

Table of Contents

Introduction

Let's face the facts. JavaScript is an essential tool to have in your toolbox. Even if you create your HTML pages with an application that creates scripts for you, there will come a time when you need a script that can't be automatically generated. There also might come a time when you want to modify one of those scripts. Most importantly, companies look to hire developers with a diverse toolset. In today's challenging job market, this can be essential to your success.

What this book does, and who it's for.

This book is for **anyone** with an interest in developing their Javascript skills, the book uses very clear examples that enable you to master the programming language. It's also a useful reference for developers.

Please bear in mind that as websites are by nature populated with ever changing content, and are also often transient and quickly redesigned, many screenshots may not exactly match those seen if the link to the site is followed.

About Tom Dell'Aringa

Tom is our **JavaScript guru**; he wants to teach you how to correctly use JavaScript by giving the basics of scripting first, then covering some simple, commonly used scripting next. Then he will move on with topics like always making sure that the code is efficient and that the user experience is good.

During the early 1990's, Tom (http://www.pixelmech.com/) spent his days as a graphic designer for a small publishing house. His boss threw a World Wide Web book at him one day, "encouraging" him to research it. Tom fell in love and quickly made the transition from print to web technologies, teaching himself along the way. During the wild ride of the dot.com era, Tom worked for a start up, web integrator Scient and a failed small design firm that bounced his paychecks. Forever fascinated by Peanuts comic strips, animation and history, he recently completed his first mini 3d film. He holds a B.A. in Fine Art from Columbia College, Chicago.

Prologue: Becoming a master Developer

Tom shows you how to use JavaScript by starting with the basics first. After that he will move on with forms, one of the trickiest javascript applets to create for starting developers. After that he moves on to real world applications such as a Javascript news ticker and a date object.

Basic Ingredients – JavaScript for Beginners

This chapter shows you how to write good JavaScript and teaches you the basics that you'll need to build your basic websites and applications. It talks about issues faced in the everyday world.

- **JavaScript Zero to Hero**
  The first paragraph covers essential principles such as writing readable and self-describing code, using correct syntax, when to use inline and linked code, what are parameters and arguments, using functions, variable scope, expressions, operators, and arrays.
- **How to Debug JavaScript** With the right tools and techniques, tracking down your errors shouldn't be too difficult. This paragraph shows you to debug your scripts and how to prevent bugs by improving your code by commenting your code smartly, writing good variable names and proper testing.
- **Two dimensional arrays** explains how I'm going to use window onload with an anonymous function and using the DOM to build a new SELECT form element.

Objects

This chapter shows you the reason you want to take an object based approach is the power it offers.

- **Building a JavaScript Object** shows you how to use the basic principles of object oriented programming such as; properties, methods and constructors.
- **Scripting The Select Object** explains how to get at the select with a script, and he'll write a useful script that you can add to your script library.
- **Scripting the Select: Moving Things Around** Often it isn't enough simply to be able to grab which OPTION the user has selected. What if you have a form where the user should be able to add or remove a value to the menu? Is it possible to actually change the values in the menu on-the-fly? What if we wanted to move a value up or down? Tom shows you how to do it.
- **Scripting the Select; Going further** Tom continues to extend the functionality to make the tool more useful. In the process, you'll learn how to further manipulate the OPTION array within the SELECT object.
- **Scripting the Select: Finishing Up** describes how to accomplish are delete a bookmark, and select all the bookmarks.
- **The Date Object** Sooner or later you're going to find yourself in need of some kind of date in your scripts and in this paragraph Tom shows you how to create such an object.

Forms

Tom shows you the power that JavaScript puts at your fingertips with tools like the elements[] array and the type properties. Correctly using these tools will make your life much easier.

- **Scripting Forms** Tom will not only show you how to script forms but also gives an explanation on how to use the DOM.
- **JavaScripting usable forms** shows you how to copy information in a form to prevent users to fill in the same information twice.
- **JavaScript: Disabling and Enabling Form Fields Dynamically** Sometimes you have a form that has dependencies, meaning one (or more) field's values depend on another field's values. For example, you may not want the user to fill in field B until the user has filled in field A. Tom shows you how to build these forms.

Advanced Forms

This chapter shows you how to extend and improve your usability and accessibility of your forms.

- **Multi Page Forms on One Page** Quite often you are faced with the prospect of a very long form. Tom shows you how to use JavaScript to make the form appear as if we were moving through a series of steps (to keep the form manageable for user) while never leaving the page.
- **Validating Forms: A JavaScript Validation Class** shows you how to build a JavaScript class (an object) with validation methods that you could reuse again and again.
- **Email address and password validation** extents on the previous paragraph by adding a new validation function to the form.
- **Formatting User Form Data** Tom takes a look at 7 and 10 digit phone numbers and how you can format (or mask) them to look nicer. While we're doing he introduces some handy string methods and talks about a very basic regular expression.
- **Scripting Radio Buttons without tears** Scripting a checkbox is relatively straightforward. Radio buttons like to throw a wrench into things. Don't let that deter you. In the end if you simply understand what a radio button really is, it will be a breeze to write those scripts you've been itching to write.
- **Scripting Checkboxes** In this paragraph we will look at groups of checkboxes and how to manipulate them within a form.

Taking it further with DHTML

This chapter describes the more advanced techniques of Javascript such as;

- **Modifying page elements on the fly** describes how to achieve separating the presentation code from the content itself.
- **Working with the Event Listener** shows you what an event is, how to use event handlers and how to work with event listners.
- **Spicing up Data Tables with Highlighting Rows** Tom builds a script that will allow your table rows to "highlight" as the user mouses over the row. Tom does this using the DOM.

Real-World JavaScript

This chapter shows you some real world applications for Javascript.

- **Cookie Handling** Tom shows you how to solve the problems that stateless pages have. In other words, pages that live in their own little world that doesn't know about any other pages on any other site, or even pages on its own site.
- **The JavaScript Date Picker** Typing in dates manually can often be a problem, especially with a global medium like the web; Americans write 3rd January 1/3; Europeans write 3/1. Are your dates 03/01/04? 3/1/04? 1/3/04? 1/3/2004? A date picker, when a user clicks on a date from a calendar-like interface can smooth the process.
- **The JavaScript News Ticker** can be a useful little tool to draw attention to the latest headlines.

Basic Ingredients – JavaScript for Beginners

JavaScript Zero to Hero

Introduction

In this chapter I'll cover some key aspects of writing good Javascript, and then move on to clearing up some troublesome issues programmers face. From there I will cover basic scripting issues faced in the everyday world. For these paragraphs I am assuming you understand the basics of Javascript, such as why you might use it, what a variable is, and what a function is. If you don't, you should take a look at W3Schools introduction at http://www.w3schools.com/js/js_intro.asp.

Building anything requires a solid foundation or it will not stand. This principle is especially true in programming. If you start off with solid building blocks, additional and more complex work becomes that much easier. If you start off with a shaky foundation, you can quickly become bogged down and further progress can seem near impossible. With this in mind, I want to cover 9 essential principles:

- Writing readable and self-describing code
- Using correct syntax
- When to use inline and linked code
- What are parameters and arguments
- Using functions
- Variable scope
- Expressions
- Operators
- Arrays

Writing readable and self describing code

The longer you are a developer the more you will find yourself faced with this situation: You open a page that you wrote 6 months or a year ago and you have to modify some code. Problem is, you have no recollection of what you were thinking back then, and you cannot remember why you did half of what you did.
Even worse, you've been handed somebody else's code with which you need to work, and you can't understand what they were trying to do either. That's why it is so important to write readable and self-describing code. But what does that mean?

Readable code means that you can read it somewhat like English. Let's take the following example:

```
t = p * r;
```

What does that code block mean? I have no idea. Contrast that with the following:

```
totalPrice = price * taxRate;
```

Now you don't have to remember that t was equal to totalPrice, because you simply read it. The first principle in writing readable code is to **write descriptive variable names**. Some good examples:

```
myForm;
currentObject;
textFieldSize;
```

Note, it's a generally accepted convention that variable names in all upper case with underscores are constants (a constant is a variable that is set once and typically never – or rarely - changes, such as a tax rate) such as TAX_RATE.
This goes a long way to help you understand what a code block you are reading is doing. The same principle goes for functions. **Write descriptive function names.** Some good examples:

```
ToggleFormField()
SwapImage()
AddTableRow()
```

Also note, it's a good idea to **have your functions start with an upper case letter and your variables a lower case letter**. This allows you to easily distinguish between a variable and a function name. This may not seem important, especially with small functions, but it can be quite helpful in more complex scripting. It's a good idea to get into this convention right from the start; it gets you straight into a best practice. If a script you are writing ends up being more complex than you thought, there will be no confusion as to what is a variable and what is a function.
Lastly, its very important to **thoroughly comment your code**. Write good descriptive comments that explain what the function should do, what parameters it takes and the expected result. This is invaluable for you (or the next person) to understand what is going on 6 months from now.

Example:

```
// Get Total Price Including Sales Tax      by Your Name  04/05/03
// This function figures out the total price of a product
// including sales tax. It doesn’t return a value, but
// places the total in the “price” field in our form.
// Send the function the base price as an integer.

function TotalPriceWithTax(basePrice)
{
     // .0815 is the Pretend County tax rate
     var taxAmount = basePrice * .0815;
     var totalPrice = basePrice + taxAmount;

     document.forms[‘myForm’].elements[‘price’].value = totalPrice;
}
```

There are other ways to comment well, but you get the idea. There will never be any doubt in a programmer’s mind what the above function should do and how it works. The comment above the function is clear and gives all that is needed to know to be utilized. Note the simple comment within the code that explains what the number means. It’s also helpful to include your name and the date the function was written, in case somebody should wish to contact you with questions.

Function Brackets

There are two generally accepted ways to use brackets with your functions. Remember, a function must have an opening and closing bracket, and each control structure (such as an if statement) must have a complete set of brackets as well. The first accepted way is:

```
function FunctionName() {

     //some code

if(){
 //some code
}
}

The second way:
function FunctionName()
{
     //some code

if()
{
//some code
}
}
```

My personal preference is the second. It always has seemed to me that things just line up better that way. It’s easy to see where a bracket group starts and ends, and whether or not you’ve missed one.

One missed bracket can easily put you in a world of hurt. Also notice the indenting as well; each level of your code should be indented.
Some people consider some of this to be additional space used that adds to download times. Frankly, this simply isn't true unless you have an extremely large Javascript page. Even then, you are probably only looking at a few kilobytes worth of data. The benefits you gain in readability far, far outweigh the few kilobytes you might have saved.

Use Correct Syntax

Any language has a syntax that should be followed, and Javascript is no exception. Straying from the expected syntax usually causes errors. In some cases you won't get an error, and this is actually worse since when it causes a problem down the road it becomes very difficult to track.
There is a syntax for almost everything you do in a script, from how it is included to how you name your variables. The best thing to do is have a handy reference available to you to check against when you are not sure. As you script more you will need it less and less, but a good reference goes a long way toward keeping you on the right track. My favorite is Danny Goodman's Javascript Bible, 4th Edition. Let's look at a couple of examples.

Javascript is case-sensitive. That means if you name your variable `cookTime` and then attempt to retrieve it with `cooktime`, you will probably get an "undefined" error. Because `cooktime` does not exist, only `cookTime` does. One that often trips up beginners is quoting strings – particularly strings inside strings. You must alternate between single and double quotes if you are going to nest strings. This is because Javascript will look at the first occurrence of the first type of quote you use, and the next occurrence as the beginning and end of your string. Examples:

```
myString = “this is some text”;
```

The string is of course between the two double quotes. However, if you did this:

```
myString = “this is some text’;
```

Javascript would not understand – even with the semicolon present – that the single quote was the end of your string. Therefore, when the next statement begins, it will be included as part of what it thinks is now a nested string starting with the semicolon, and the inevitable errors follow.
To nest a string, you would do:

```
myString = “John said ‘this is fun’”;
```

Note 'this is fun' is single quoted within a set of double quotes, and so is correctly parsed.
Tip: Generally people get used to using double quotes for their strings. It actually makes things a lot easier to begin with single quotes. When you get into more complex coding, such as writing HTML inside strings, it makes it easier to do, as the following example shows:

```
myHTML = ‘<form name=”myForm” method=”post”>’;
```

Notice how this allows me to use double quotes for my HTML element attributes without *escaping* them. Escaping characters means prefixing them with a \ so Javascript knows to ignore them. Had I began with double quotes, I would have had to do:

```
myHTML = “<form name=\”myForm” method=\”post”>”;
```

The first example is easier on the eyes and less work. This is a simple example; it makes even more sense with more complex code.
Obviously, there is a lot more syntax than what I have described. I’ve highlighted a couple common issues that arise and can bite you. The point to remember is to use your handy reference if you have any question about how something should be written.

Inline and linked code

There are a few ways you can include Javascript in your pages. While you can put a script block anywhere within your page, that doesn’t mean you should do so. The most common case you see is local page scripts in the head of the document:

```
<head>
     <title>Page Title</title>
     <script language=”Javascript” type=”text/javascript”>
     // javascript code here
     </script>
</head>
```

This is perfectly acceptable. Note: you should always include the language and type attributes in your script tag. Do not leave them out. However, you should be thinking about the scope of the code on your page. Could these functions be used elsewhere on your website or application? You certainly never want to reproduce scripts on multiple pages.
I only put Javascript code locally in my page when it is absolutely needed. That means that particular page needs that particular code, and it is only used there and nowhere else. It is unique to that page. Otherwise, it’s a good idea to simply link to a file that has only your Javascript in it like this:

```
<script language="JavaScript" src="/scripts/common.js"></script>
```

This way you have one central location to edit your scripts across all your pages that include the file. In addition, the script will be cached by the browser so it only needs to be loaded once.

Parameters and Arguments

This concept often confused me when I was a beginner. The first thing to remember is that parameters and arguments mean the same thing. Secondly, parameters and arguments are sent to objects that act upon them. We’ll stick to the term “arguments” to make things simple.
Think of a candy machine. A typical candy machine might take two arguments: money and a selection value. The machine cannot really act successfully unless it gets both of these things. So you supply the two arguments: you put in sixty-five cents, and you press “D5.” The candy machine responds to those two arguments by turning the appropriate spindle and dropping a Milky Way down the chute.

The same thing is true of a function. Remember the TotalPriceWithTax() function we referred to above? It took one argument – basePrice. What usually confuses people is the name of the argument and the thought "where did that come from?"

Look again at the function:

```
function TotalPriceWithTax(basePrice)
{
     // .0815 is the Pretend County tax rate
     var taxAmount = basePrice * .0815;
     var totalPrice = basePrice + taxAmount;

     document.forms['myForm'].elements['price'].value = totalPrice;
}
```

The answers are simple. First, a descriptive name and good commenting will tell you what the name of the argument means. In this case, it's the base price of some item. But what is its value? Its value is sent when the function is called. If we called this function using an onClick event, it should look like this:

```
onClick = "TotalPriceWithTax('4.99');"
```

Now, hopefully you see that *in this instance*, the value of basePrice as the argument is 4.99. Why? Because the function takes that value – 4.99 in this case – and assigns it to the basePrice variable. You could call this function any number of times you wanted to, sending a different value for a different item each time, where the price is unique to that item. Each time, the function sets the variable basePrice to the number that was sent during the function call.
Now the variable basePrice can be used anywhere in the function you want, and its value (in this case) is 4.99. So that is acted upon in the following line:
taxAmount = basePrice * .0815;

In our example, this line means the same as:

```
taxAmount = 4.99 * .0815;
```

Without getting into programming theory, using arguments helps you to reuse your functions over and over. I can use this function to get the total price of ALL my products, regardless of how many I have. You would never want to write a function like this for only one product. If you had 500 products, you would need 500 functions!

Using functions

How do you use functions? In general, most of the time functions are called or referenced using event handlers. Depending on the object you are using (form, button, anchor, etc.), you have different event handlers available to use. For example, a button has an onClick event handler. A click is an 'event' and when the button is clicked, it generates an onClick handler.
You can use this to run your functions. Just as we did above when we called our example function. The whole tag would look like this:

```
<input type=”button” name=”buttonName” value=”Get Total Price” onClick=
“TotalPriceWithTax(‘4.99’);” />
```

(Note: this is only a simple example, but you may have noticed a problem. Did you see it? If we have more than one product, we would have to change our function call each time since we are passing the price of a single product to our function. We’ll get to more advanced concepts in later chapters, but what you would really want to do is pass the function a *reference* to your *current product’s* price. That might mean passing the value of a price field to the function. Or it might mean that some server-side scripting language (such as PHP or ASP) dynamically writes in the current product’s price based on some earlier action – such as selecting a product to view).
Functions can also be called *within* functions. Let’s say we wanted to round off the price we end up with in the first function. We could simply add that code to `TotalPriceWithTax()`. But if we did, we could never use that code anywhere else. It would have to be rewritten again.
If we wrote a function that rounded off that number called `RoundPrice()`, we could use that function anytime we wanted to perform that action without rewriting it. Then, you can call `RoundPrice()` function within `TotalPriceWithTax()` like this:
function TotalPriceWithTax(basePrice)

```
{
     // .0815 is the Pretend County tax rate
     var taxAmount = basePrice * .0815;
     var totalPrice = basePrice + taxAmount;

     roundedTotalPrice = RoundPrice(totalPrice);

     document.forms[‘myForm’].elements[‘price’].value = roundedTotalPrice;
}
```

Making sure that your `RoundPrice()` returns the new, rounded value, we can assign the new function to our `totalPrice` variable. This means the new rounded value is put back into the `totalPrice` variable, and outputted to our price field just as before.
You can also call functions from an anchor tag. There are right ways and wrong ways to do this that have to do with accessibility and usability. We’ll save that for another time.

Variable scope

Scope means “where is it recognized.” Any variable you set *within* a function (inside your brackets) is said to have **local scope**. That means if you try to refer to one of those variable outside the function, it will throw an undefined error. Only the function knows what that variable is. So in our function example above, the variable `totalPrice` cannot be used outside the function, as it will never have a value.
If you define a variable outside a function, it becomes, in a sense, a global variable. It becomes available to any function or code block **in the same page**. Be careful how you use the term global, however. It is global to the particular page in which it resides. Remember HTML is stateless. Each time a page is loaded everything starts over.

Therefore, if we had some code like this:

```
var myCounter = 0;

function AddCount()
{
     myCounter = myCounter + 1;
}

function SubtractCount()
{
     myCounter = myCounter – 1;
}
```

Notice that we can use `myCounter` in both functions. Furthermore, `myCounter`'s value will keep changing and remember its value each time you change it until the page is reloaded.
Note that I didn't use the keyword "var" inside the functions when I first used the `myCounter` variable. Had I done that, it would have become a local variable within that function. The keyword "var" is used when you first declare a variable. If you were to use the var keyword again, you would re-declare the variable, which you do not want to do. You could actually have a global and a local variable with the same name but you should try to avoid that. It can lead to confusion.

Statements and Expressions

What is a statement? Well pretty much anything you write within the script tags is a statement. Basically when you begin typing and then end with your semi-colon you've written a statement. For example:

```
Var someVariable;
myForm = document.forms['theForm'];
myCounter = myCounter + 1;
```

Those are all statements. The statement is the building block of your scripting. Expressions are first of all a type of statement, just like the rest of your code. Additionally, an expression evaluates to some value. This can mean results that are numbers, object references or even concatenated strings. The last statement above is also an expression, since it evaluated the value of `myCounter` and added 1. Some examples of expressions are:

```
theNewQuote = "He wanted a new " + "bicycle.";
scoreResult = "Your final score was " + score;
myIQ = testScore * (brainWidth – brainHeight);
```

In general terms, when you talk about an expression you expect some kind of resulting value. Statements are the real workers of your scripts. They "do" things. For example, if I wanted to change a value in my text form field, I would write the following statement:

```
var newTextValue = "super!";
document.forms['myForm'].elements['myTextField'].value = newTextValue;
```

The statement performs an action in my page, it set a value in a form element to some new value. Understanding the intricacies of what statements and expressions really are is not vital to be a good scripter. But since these words do get tossed around, I wanted to at least give you a general idea. One note about semi-colons. You may have noticed I end each statement with one. While this is not required, it is good practice. It shows you where the ending of one statement is and another begins. Also, in some other scripting languages (such as PHP) it is in fact required, so its not a bad habit to get into.

Operators

Operators are used to change or modify values. They fall into all kinds of technical sounding categories, but the main ones you will use every day make a lot of sense. To compare things you use comparison operators, such as:

```
> (is greater than), < (is less than) and == (equals)
```

You have your basic operators you learned in school like:

```
+ (plus), * (multiply) or even % (modulo, or remainder), plus some neater ones
like ++ (increment) and -- (decrement)
```

Assignment operators allow you to set up variables and references, some of which you already know. Using the = sign you assign a variable. For concatenating you can use the nifty:

```
+= (add by value) such as myCounter += 1 would add 1 to the myCounter variable.
```

Boolean operators are used for “and” and “or” comparisons. If you wanted so say “if Sally and Harry” you could use the *boolean and* like this:

```
If(sally && harry)
```

You can also use two pipes - || - for *boolean or*, and even a ! for not. Again, a handy reference is always helpful to make sure you are using the correct operator.

Arrays

One thing it took me a long time to learn and make use of is the fact that Javascript makes use of arrays in quite a few places to form collections. You can also make your own arrays as well, but the arrays I want to point out are the ones that are part of the Javascript core language. These are available for your scripting use and should be used when appropriate.
For example, when a page is done loading, there is an array set up that includes a reference to every single image on the page. Each image can be referenced by its location in the array via its array position (the order it loaded on the page – and remember Javascript arrays start with 0 not 1) or its name attribute. So if the first image on my page is this:

```
<img name="westLighthouse" width="10" height="10" />
```

I could access it two ways:

```
myImage = document.images[0]; or
myImage = document.images['westLighthouse'];
```

The same is true of every form, form element, anchor, link and each option in a select element. Using these arrays is a smart and powerful way to script. Generally using the name attribute, not the array position, is the correct way to use the array. Not only does this safeguard your reference if things on the page move around, but it goes along with our self-describing code guideline. "westLighthouse" has much more meaning than "3."
This is especially helpful in scripting forms. The best way to access a text field named "myText" in a form called "myForm" would be:

```
someText = document.forms['myForm'].elements['myText'];
```

In the above case, we use the forms array to get our form (myForm) and the elements array of that form to get the text field (myText).
Not only is this easily readable and understood, this is the way the original DOM (Document Object Model) described these elements. That means when you use this code it's valid in the most modern browsers all the way back to Netscape 2 and Internet Explorer 3! The less you have to worry about cross-browser code, the better.

Conclusion

Remember, with a strong foundation you can build strong scripts. Write clear and self-describing code (with lots of comments) and use the correct syntax. You'll be a happy scripter when you do.

How to Debug JavaScript

If there is one email I get more than any other it is "this script doesn't work." Now, granted I'm certainly not infallible but I do take care to make sure I publish a working script! In fact, I'll even tell you that every script I have ever posted here as worked. The problem is that sometimes the **presentation** of that script has occasionally left some ambiguity that can be confusing.

Invariably what happens is the script breaks. Now the person immediately wants to email the author (me) saying something to the effect of "I copied your script and it's busted man!" (Okay they are usually a little more tactful that that.)

So being the good Samaritan that I am (cough) I grab their URL/script/e-mail and begin debugging. And this is the first problem! Before you ask someone a question about a script you got from them – and more importantly – before you start asking questions on forums about scripts that YOU wrote, you should be vigorously debugging them.

Because you don't want to waste the people's time in the forums with obvious issues you could have corrected first (this getting to the real issue – or realizing there is no issue at all and it was a simple error). When you get the reputation on a forum, nobody wants to answer your questions, and you don't want that! And you certainly wouldn't want to waste the authors' time right?! ☺

Proofreading

The first thing you should do, the thing that is the easiest to overlook, is to proofread your script. What I mean by this is that you are looking for the OBVIOUS errors that stick out like a sore thumb – and the things that aren't so obvious but that you might see once you look it over. It can be easy to miss things like this when you are busy writing a script. You might have meant to go back to something and forgotten. So even before you fire up your script, give it a read through. That will allow you to catch things like:

```
if(10==11
{
     Var foo=1;
}
```

or

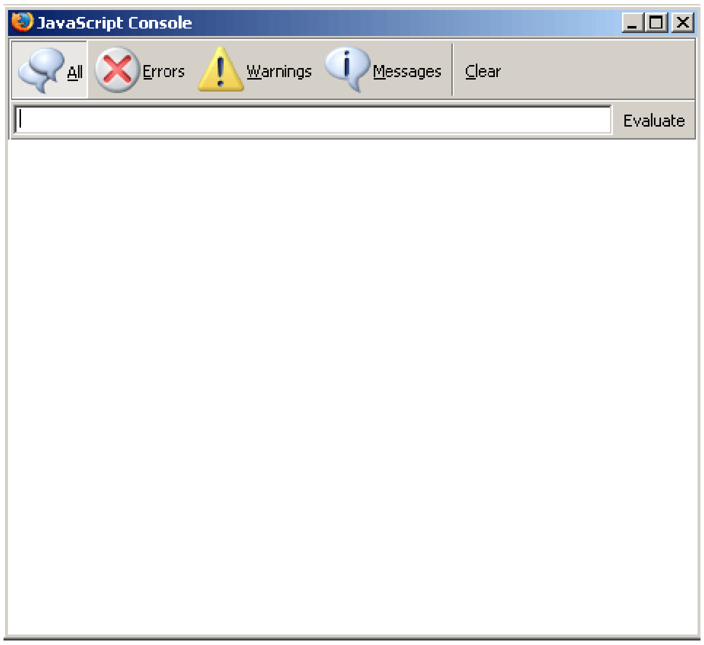
```
// this is a comm
ent

x = 27;
```

What are the two errors above? The more you work with scripts the better you will get at spotting errors, so it can be tricky at first. In the first example the **if** statement is missing the closing parenthesis.

In the second example, the comment has wrapped to the next line, so the characters "ent" are simply "floating" in the script. This is in fact a situation that I got more than one email on. These things should be caught either by proofreading or by a JavaScript Console.

What JavaScript Console?

If you aren't using the JavaScript console, then you are subjecting yourself to unwanted and unneeded pain! Wouldn't it be great if there was a tool that would tell you what your error was and where that error was? Sure I would!

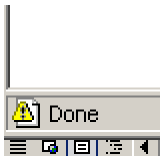


Ooh, now does that look sweet or what? Where did I find this little gem, you ask? What you are looking at is the JavaScript Console in the FireFox browser. You get it by going to **Tools** > **JavaScript Console**. Eh what? You don't use FireFox you say? (Shame on you!). Alternatively, there's the Netscape 7 console located under **Tools** > **Web Development** > **JavaScript Console**. What about Internet Explorer, you ask? Nope, sorry. Download one of the two browsers above if you don't have them already.

What about the Microsoft Script Debugger?

Sure, there's always some smart aleck in the back who has to ask that question. Sure, you can TRY to use that if you are a masochist. The script debugger is a tricky tool to use at best. Furthermore it is not nearly as effective for most people as the JavaScript Console. Enough said!

Now, IE does have a script error notifier. It can be turned on and off in the preferences, and is usually turned off by default. IE will tell you there is a script error by the lower left hand corner 'alert' indicator like so:



Note the little yield sign. Double clicking that will show the error. The problem is this version is much less reliable and friendly than the console, which is why we stick with that. You can still get useful information from IE in this manner, but the console is simply more reliable and will give you better and more accurate information.

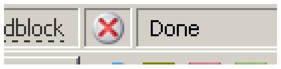
We'll stick with FireFox, since that is what I use and find easiest to get to (not as buried in the navigation as the Netscape one is.)

Debugging in the Console

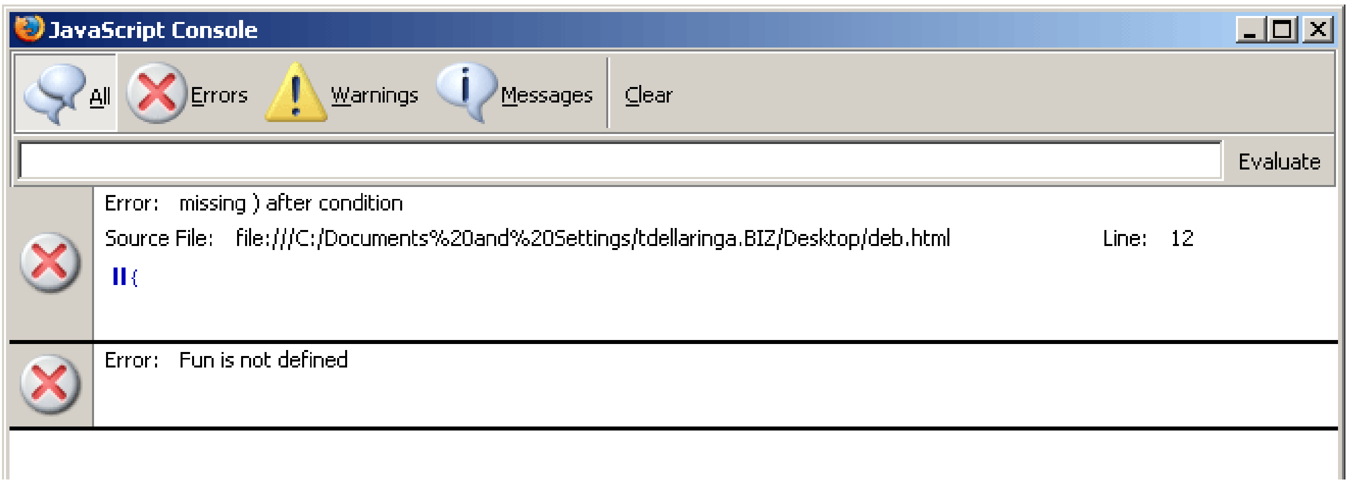
Let's take the errors we showed above and put them in a test script like this:

```
function Fun()
{
     if(10==11
     {
           Var foo=1;
     }

     // this is a comm
     ent

     x = 27;
}
```

If we call this function using the **onload** event handler, here is what we will see in the BROWSER:



In the lower left hand corner, that little red 'x' tells you there is a JavaScript error. So you are sitting there wondering why your blinking, scrolling, pop-up marquee isn't showing up, glance down and look for the red 'x'. If you double-click it, you'll get the console up showing you the following:



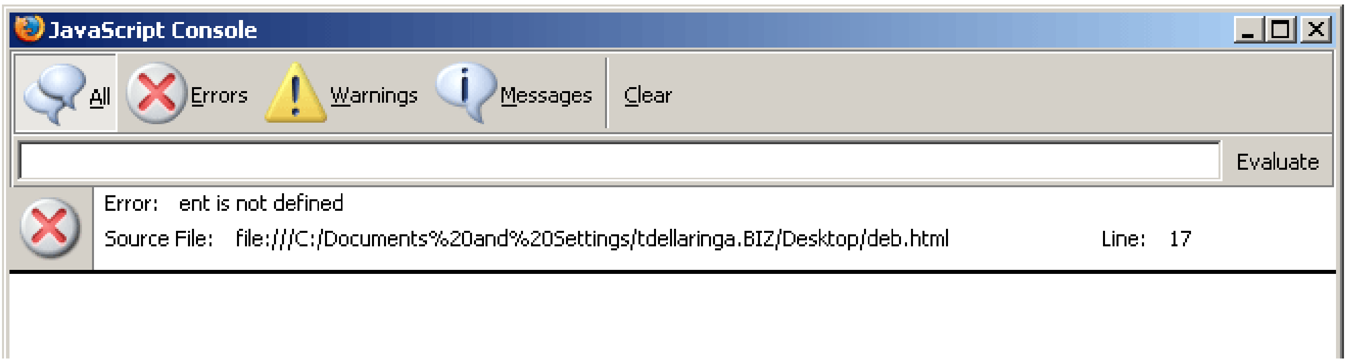
```
Error:   missing ) after condition
Source File:   file:///C:/Documents%20and%20Settings/tdellaringa.BIZ/Desktop/deb.html     Line:  12
||{

Error:   Fun is not defined
```

Note we have 2 errors, but you have to be careful. The first error sometimes throws EVERYTHING off after it. In this case since the missing **)** causes everything to be interpreted differently, that is what happens. In any event, notice how the exact error has been stated for you:

```
Error: missing ) after condition
```

It even tells you the line number it is on. One caveat with line numbers, however. If the script is local to the page you are looking on, the script line should be dead on. If it's in a remote file linked with a SCRIPT tag, it may or may not be the right line. That's okay though, you've got the exact information you need.

You look at your script, slap yourself in the head and fix the error. You rerun the script and get the same red 'x'! So you check again and see:



```
Error:   ent is not defined
Source File:   file:///C:/Documents%20and%20Settings/tdellaringa.BIZ/Desktop/deb.html     Line:  17
```

It is telling you that 'ent' is not defined. What is 'ent'? Exactly – it's nothing:

```
// this is a comm
ent
```

Since the comment wrapped, the browser thinks 'ent' is a JavaScript statement. This is quite easy to fix.

The console will be very specific about many errors in this fashion, and will make your life about 900% easier.

Common Errors

Hopefully you won't make those kind of errors very often. Those are called 'syntactical' errors. You have written JavaScript incorrectly, or out of syntax. The console is excellent for those types of errors. But there are other types of errors. Here's a few that crop up quite often it seems:

Object Expected or Undefined

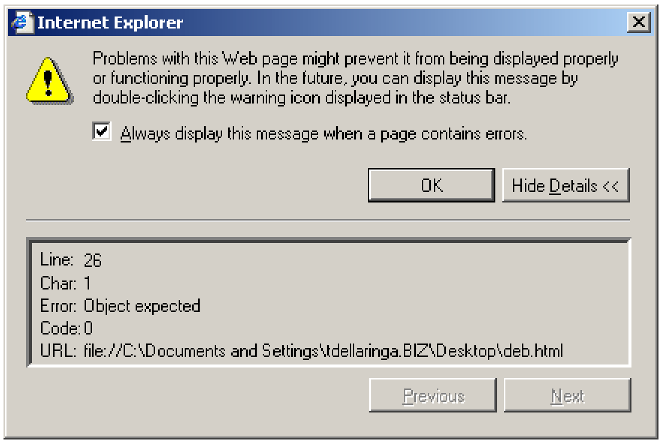
This should be your clue that you've forgotten something. You've forgotten to include a link to your script or completely failed to put the script in your page. You might have misspelled the function name. Basically what we are saying here is JavaScript is not finding the script. Example:

```
function George()
{
   … statements
}
```

And the function call:

```
<body onload="george();">..
```

IE will complain of "object expected":

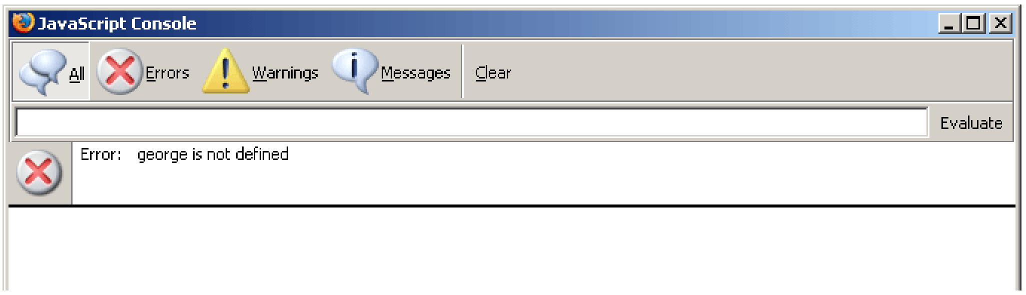


Problems with this Web page might prevent it from being displayed properly or functioning properly. In the future, you can display this message by double-clicking the warning icon displayed in the status bar.

☑ Always display this message when a page contains errors.

```
Line:  26
Char:  1
Error:  Object expected
Code: 0
URL:  file://C:\Documents and Settings\tdellaringa.BIZ\Desktop\deb.html
```

In this case the line number is where the function call was made, although IE line numbers are notorious for being unreliable. The console gives you a better description, "Error: george is not defined":



Error: george is not defined

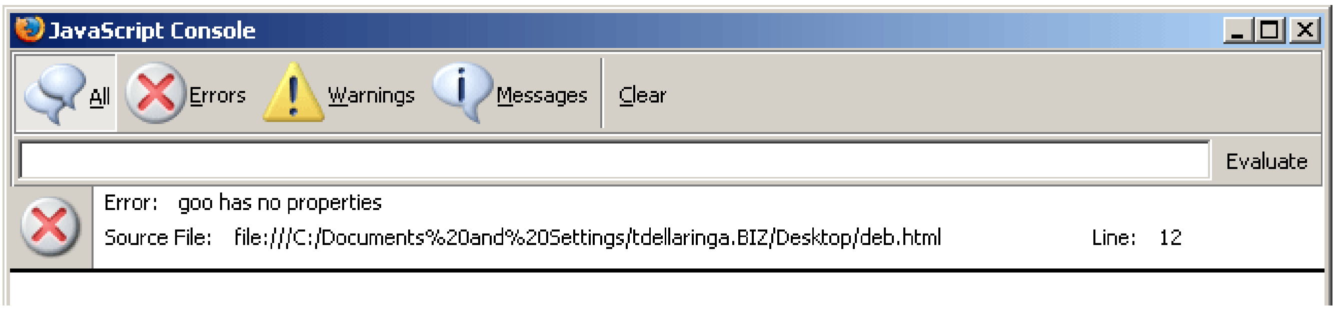
Wait, we *do* have a function named george right? Wrong – it's "George" – capital G. So in fact this error message is indeed accurate – george is not defined, even though George is.

Object has no properties

This one can crop up and really confuse you. What happens is you are generally trying to point to an object that isn't there. You forgot to create it, you misspelled it or it is out of scope. Take the following silly script:

```
function Fun()
{
     var goo = document.getElementById('nothingHere');
     var foo = goo.value

}
```

Assume that goo is something you have to deal with in your script. But the element 'nothingHere' isn't in your HTML for whatever reason. When you attempt to get the value of goo you get:



Error: goo has no properties
Source File: file:///C:/Documents%20and%20Settings/tdellaringa.BIZ/Desktop/deb.html Line: 12

Goo cannot have any properties because it doesn't exist! Note that in this case IE still gives you the "object required" error.

You'd be surprised how many errors are variations on this theme.

Other common errors

There are some things that crop up all the time even among seasoned scripters. Mismatched quotes:

```
var string = "John said 'Hi there';
```

The missing end quote here will cause you lots of fun trouble. Always scan for correct use of quotes. A very easy one to do is to use a single equal sign instead of a double when testing for equality like so:

```
if(x = 7)
{
     statements…
}
```

This of course won't check for anything and you will never enter your conditional statements. The correct statement is x **==** 7.

Calling scripts before the page is loaded.

If you attempt to run a script that uses an object in the page, but that page isn't yet loaded fully, you'll get the **object expected error**. Make sure your scripts are loaded *after* the page loads by either using the **onload** event of the BODY tag or using events within the page itself (like a button click) to call the function.

Logic Errors

The last type of error is the logic error. A logic error is when you have made an error in your...logic. You see, a computer only does what you tell it to do. If you tell it wrong, it does what you tell it to do. If there isn't an obvious syntax error connected with the issue then it is likely that nothing will happen at all – including what you were expecting.

This is where the alert becomes your friend. You need to see what is exactly happening in your code because the console is not going to tell you, since there is not really an "error" per se, but more of a miscalculation on your part. It is really a "human" error.

What you want to do when this occurs is to place alerts within your code to check your values, and see if they are what you expect.

Imagine you have this function:

```
function Fun()
{
     var x = 100;
     var y = document.getElementById('y').value;

     var z = y - x;

     switch(z)
     {
           case 25:
           statements...
           break;

           case 30:
           statements...
     }
}
```

You have the value for **x** which is 100, and then you are pulling a value from an element in your page named 'y'. You expect the result of this to be either 25 or 30 and write a switch statement for that but it never executes. You aren't going to get an error here since **y** is indeed a valid element with a value. But apparently that value is not what you expect.

The best thing to do would be to place some alerts and check the values of y and z. When you see what those values actually are, you will probably see why your result isn't 25 or 30 as you expected.

So when you have no error indicator but the script isn't working, go after the logic. Place some alerts and see what the script is actually doing. Sometimes it also helps to watch somebody go through a page with your script on it, they may do something differently than you were doing that will tip you off.

Conclusion

With the right tools and techniques, tracking down your errors shouldn't be too difficult. Really writing good scripts is harder than tracking down the errors. Things that help you out are commenting your code smartly, writing good variable names and lots and lots of testing!

Two Dimensional Arrays and other Goodies

Quite often the secret to writing good scripts is knowing enough "trade secrets" to make a function really scream and work well. It seems like over time you learn more and more tricks of the trade and realize how you could have done things better. This in turn makes you a better scripter.

I'm going to give you a few tricks of the trade to use involving three things: Two dimensional arrays, using **window.onload** with an anonymous function and using the DOM to build a new SELECT form element. Sound obscure? I've used ... er, I mean a **friend of mine** has used these techniques for loading advertising banners , and using a script to randomly select one of the array items in the first column, which is the banner, then grabs the associated link from the second column. Anything along those lines it would be great for.

The Script

The script that will hold our dirty little secrets might or might not be useful in the real world. The script we will build will essentially be a poor-man's constructor to build a SELECT form element. I once worked in a real world situation where every time you wanted to add a form element you had to use custom JavaScript objects similar to this script to make them. It was quite tedious in many ways. However it did give them absolute control over the elements. They knew that if their programmers used the methods they set forth that their application would be standard across the board. This script could be used in a similar fashion if one wanted to do so.

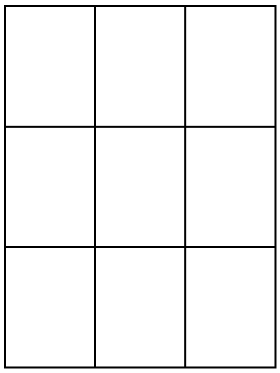
The script will work by taking a two dimensional array as a parameter and outputting a SELECT element via the DOM. But what is a two dimensional array?

Two Dimensional Arrays

JavaScript doesn't support true multi-dimensional arrays like some fully featured languages use (like Java for example). A real multi-dimensional array would allow you to specify the "matrix" right off the bat, like so:

```
myArray = new int[3][3]; //a true Java multi-dimensional array
```

This would give you an empty "matrix" like this:



Which you could then fill with data. However JavaScript doesn't allow this type of construct. What it DOES allow is a mocked up version called a two-dimensional (or three or four) array. In truth, it's really an *array of arrays*.

Array of Arrays?

What we're going to need for our script is a matrix that is 2 x 3, so let's we'll begin there. Remember in JavaScript to create a new array you can do the following:

```
var myArray = new Array();
```

And then give it values:

```
myArray[0] = "foo";
myArray[1] = "bar";
```

And so on. You could also do it a bit shorter:

```
var myArray = new Array("foo", "bar");
```

It depends on how many members of the array you are working with and how you like your code to look. I prefer the first method because it seems cleaner. At any rate, we want an array of arrays. The sweet thing is that the members of an array can actually BE arrays! So, if we want a 2 x 3 array, we can set it up. First we need the "2" part:

```
var myArray = new Array();
arr[0] = new Array();
arr[1] = new Array();
```

We now have an array that holds TWO arrays as its members, Like this:

array #1	array #2

Both of these arrays are empty at the moment, but you can see that we have the start of a matrix. If we populate those arrays, the matrix will expand, so let's do that.

Normally, we populate an array like we did above:

```
myArray[0] = "foo";
```

In this case, we are using a two-dimensional array, so we have to specify which column and row the data goes into. Think of it as working with table data, it's really the same concept.

```
var teamArray = new Array();
teamArray[0] = new Array();
teamArray[1] = new Array();

// these are for VALUE attributes of the OPTION
teamArray[0][0] = "south";
teamArray[0][1] = "north";
teamArray[0][2] = "none";

// these are for the TEXT attributes of the OPTION
teamArray[1][0] = "White Sox";
teamArray[1][1] = "Cubs";
teamArray[1][2] = "Neither!";
```

Notice the double brackets. Visualize those brackets as if the first bracket told you what column to look in, and the second bracket told you what row. Like so:

[0]	[1]
south	White Sox
north	Cubs
none	Neither!

Now we have built our matrix! You'll notice via the comments that the first column will be used for the VALUE attribute of the OPTION and the second column for the TEXT that shows up in the browser.

Now we need a function to use this wonderful data.

The Function

I'm going to throw the whole kit and caboodle at you and then explain it:

```
var teamArray = new Array();
teamArray[0] = new Array();
teamArray[1] = new Array();

// these are for VALUE attributes of the OPTION
teamArray[0][0] = "south";
teamArray[0][1] = "north";
teamArray[0][2] = "none";

// these are for the TEXT attributes of the OPTION
teamArray[1][0] = "White Sox";
teamArray[1][1] = "Cubs";
teamArray[1][2] = "Neither!";

function BuildSelect(arrayOptions)
{
     var arrayLength = arrayOptions[0].length;
     var oForm = document.forms['baseballForm'];
     var newSelect = document.createElement("SELECT");
     newSelect.className = "ourSelect";
     option = document.createElement("OPTION");
     newSelect.appendChild(option);

     option.value = "";
     option.text = "Choose your team!";

     for(var i=0; i<arrayLength; i++)
     {
           option = document.createElement("OPTION");
           newSelect.appendChild(option);

           option.value = arrayOptions[0][i];
           option.text = arrayOptions[1][i];
     }
     oForm.appendChild(newSelect);
}
```

You'll notice that our function only takes one parameter, and that is our two-dimensional array which we named arrayOptions. Basically, we're feeding our function with our arrays of arrays, that whole 2 x 3 matrix of data!

Note this function is going to presuppose you have an equal number of VALUE and TEXT members in each of your columns (and why wouldn't you?) So the first thing we want to know is how many options we are going to have so we can step through with a FOR loop. So we find the length of the first column (which should be identical to the second column(or array)):

```
var arrayLength = arrayOptions[0].length;
```

After setting up a reference to our form (oForm) we're going to start using the DOM to create the SELECT. Remember the wonderful document.createElement() method allows to build any element:

```
var newSelect = document.createElement("SELECT");
```

Then we can use the newSelect variable which points to our new SELECT object to attach values to its properties. Assign it a class name from our stylesheet:

```
newSelect.className = "ourSelect";
```

Our SELECT will be nothing without OPTIONs! We'll do the bulk of that work in our FOR loop, but let's toss in our "please choose" default value first:

```
option = document.createElement("OPTION");
```

Just like we created the SELECT, we now create an OPTION. Before we go doing anything with it, let's attach it to our SELECT object using appendChild(), which – yes – attaches (appends) a child to a parent:

```
newSelect.appendChild(option);
```

We can now use the familiar object.property syntax to give it a value and text:

```
option.value = "";
option.text = "Choose your team!";
```

We don't need a value for our default, and we put a simple message that the user should choose a team. Having done that to get our first OPTION, we do almost the same thing in our FOR loop:

```
for(var i=0; i<arrayLength; i++)
{
     option = document.createElement("OPTION");
     newSelect.appendChild(option);

     option.value = arrayOptions[0][i];
     option.text = arrayOptions[1][i];
}
```

As usual we are using the length of the array to determine how many times to step through the process. We happen to know it will be three times. So for each of the three times we'll first create an OPTION and append it to the newSelect.

Then we'll add the value and text properties. Only this time, we're going to use our array "matrix" values. This is where our two-dimensional array comes in quite handy. Remember the "0" column was our values and the "1" column was our text. So the first time through the loop we will get the following:

```
option.value = arrayOptions[0][0];
option.text = arrayOptions[1][0];
```

Those values, if you look at the matrix above are "south" and "White Sox." The FOR loops continues the next two times, grabbing the Cubs and none information.

Now that our SELECT is populated, we need to actually get it into our document. We'll do this by appending it to our FORM:

```
oForm.appendChild(newSelect);
```

And now the function is complete. But wait – we haven't run it yet...

Fun with window.onload

Quite often you see such functions called using the onload event handler in the body:

```
<body onload="BuildSelect(teamArray);">
```

While this is perfectly legitimate, there are other useful ways we can work this function. Sometimes it can be a pain to deal with this type of logic in the content of your page. We can instead call this function in our script:

```
window.onload = BuildSelect;
```

But wait, you say – you've forgotten the parenthesis and the arguments! Ah, there's the rub. You see if we were to call this function the normal way, the return value will be assigned immediately. This is a bad thing. Why? Because at this point (the point in the script where this line is executed) the page has not loaded and the FORM doesn't even exist! If we do it the "normal" way, the page will NOT wait for the onload event at all. We don't want to evaluate the function at this point – we only want to tell the onload handler what function to call when the onload event is complete. Leaving off the parenthesis does just that, it tells the onload handler – "hey, when you are ready, BuildSelect is your man!"

But this creates another problem – we need to call our parameter (the two-dimensional array) and this method doesn't allow that. Well that's where our old friend the anonymous function comes in.

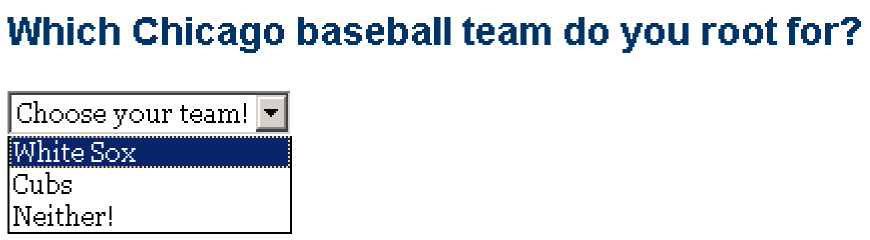
```
window.onload = function() {
     BuildSelect(teamArray)
};
```

What we've done here is tell the onload event of this window that when the onload event is complete, go ahead and execute BuildSelect(teamArray). This is exactly what we want.

Putting that all together in our page gives us the following HTML and Script:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
<head>
     <title>Untitled</title>
<style>
h1 {
     color: #036;
     font-size: 1.2em;
     font-family: Arial, sans-serif; }

.ourSelect {
     font-size: .8em;
     font-family: georgia; }
</style>
<script type="text/javascript">

var teamArray = new Array();
teamArray[0] = new Array();
teamArray[1] = new Array();

// these are for VALUE attributes of the OPTION
teamArray[0][0] = "south";
teamArray[0][1] = "north";
teamArray[0][2] = "none";

// these are for the TEXT attributes of the OPTION
teamArray[1][0] = "White Sox";
teamArray[1][1] = "Cubs";
teamArray[1][2] = "Neither!";

function BuildSelect(arrayOptions)
{
     var arrayLength = arrayOptions[0].length;
     var oForm = document.forms['baseballForm'];
     var newSelect = document.createElement("SELECT");
     newSelect.className = "ourSelect";
     option = document.createElement("OPTION");
     newSelect.appendChild(option);

     option.value = "";
     option.text = "Choose your team!";

     for(var i=0; i<arrayLength; i++)
     {
           option = document.createElement("OPTION");
           newSelect.appendChild(option);

           option.value = arrayOptions[0][i];
           option.text = arrayOptions[1][i];
     }
     oForm.appendChild(newSelect);
}

window.onload = function() {
     BuildSelect(teamArray)
};

</script>
</head>

<body>

<h1>Which Chicago baseball team do you root for?</h1>
<form name="baseballForm"></form>

</body>
</html>
```

And running this script gives you this result:


Which Chicago baseball team do you root for?

Conclusion

There are three concepts here that you should store away and use to your advantage. The first is the two (or three or four) dimensional array. These can be quite powerful in storing and handling data. Sometimes if you get a large bit of data you want to handle with a script, this type of array of arrays can be very helpful. You'll also find that in dealing with SELECT objects this type of construct can be used quite a bit.

Secondly is the DOM function document.createElement(). Today's modern browsers handle this type of operation very well as long as you stay away from the setAttribute() method. Simply stick with setting values using the object.property method like we did with option.text = "White Sox" and you will be a happy camper.

Lastly is the window.onload / anonymous function routine. Sometimes it's nice to keep your business logic out of your HTML and in your scripts, if only for ease of maintenance. The anonymous function allows you to call parameters from your function if you need to, a very handy thing to have at your disposal.

Objects

Building a JavaScript Object

Why Objects?

Power. Why confuse the issue with lots of babble? The reason you want to take an object based approach is the power it offers. We've had some requests for some form validation techniques. Some of the best techniques are based on object-oriented principles. So before we tackle that, let's take a look why JavaScript objects are powerful.

Object-oriented programming is based on the foundation that every element you interact with is an object with its own properties and methods. How does this benefit you? In many ways, from code reuse and code maintenance to scalability and extensibility. There are two main principles of object-oriented programming to take away.

First, that of **encapsulation**. What does that word mean? Taking a look at the first two dictionary entries will help out a lot:

1. To encase in or as if in a capsule.
2. To express in a brief summary; epitomize: *headlines that encapsulate the news.*

If you take a medicine capsule, you know that it will help you get better. But what you don't know is *how it works*. Even if you read the ingredients (which you do NOT have to do for the capsule to work) it still won't tell you exactly how the medicine interacts with your body to make you better.

A headline in the newspaper gives you a summary of what the story is about, but none of the details. These are both excellent examples of encapsulation.

With an object, you only know what the object is and how to use it (its properties and methods.) You don't necessarily have to know *how it works*! If you have a car object, you might have a drive and park method. You can use those methods to drive and park the car. How it happens isn't important. So encapsulating the object allows many people to use it without knowing the inner workings.

Secondly we have **inheritance**. This tells us that many objects share many common features. Sticking with the car object, there might be a Corvette, a Viper and a Beetle. They are all cars, they all drive and park. But they will have some differences. A Corvette will drive faster than a Beetle. They will be different colors and the cost will be different. But they are both cars, the same type of object.

This allows you to do less work and have more control over your code. Once you build a type of something, you can easily build many variations on the one type. For example, you could have a validation object with many types of methods. They all do validation, but they validate different types of data!

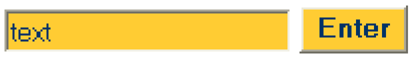
A Custom Form Object

Let's say we are working on a project that needs some kind of custom control. It doesn't even matter what it is. Let's say this control will be repeated in various places. Aha! We could use an object, instead of re-coding it each time we need it.

For the sake of brevity, let's pretend our control is simply a styled text field with a button next to it. We'll even say it's a self-contained form. So the code, if we were to write it each time we needed it, would look like this:

```
<form id="theForm">
<input type="text" id="someText" size="20" style="background: #ffcc33; color:
#003366" value="text" />
<input type="submit" value="Enter" style="background: #ffcc33; color: #003366;
font-weight: bold;" />
</form>
```

And you would see:



Think of an object as a template, or a cookie cutter. Every time you "use" it, you get the same thing. That is the case here, each time we use it, or "instantiate" it (create an "instance"), we'll get the same thing - without having to rewrite the code.

Each object will have its own properties, and usually at least one method. And, if you want - instances can even have their own unique properties.

Getting Started

Since you already know what the code needs to be, you can get started on writing the code for the object. The simple way to get started is first to write down what properties you think the object will need, and what you will need to do with the object.

What are properties?

In our example case, some properties are the size of the text field, the colors used, and maybe even the ids of the elements. You don't always want to make everything a property, but certainly the things that might need changing later on, or that you might want to customize on an instance basis should be treated as such.

So if we were to make a list, I think we would come up with this:

1. Form name (in case two controls are on one page)
2. Text field size (in case we ever need to adjust them all - remember, you can always change one by itself if needed)
3. The colors

These will become our properties.

What are methods?

What's a method? A method is simply something that tells the object to take some kind of action, or does something with the object. In our case, we at the very least need to write it out to the page. So our method list is pretty short:

1. Write object to page

"Instantiating" or "making" an object

Before we get to the code, you should be aware how to actually "make" an object from your new object code. When you get an instance of your object, its called "instantiating" an object. You create an instance. You do this by the following code:

```
var myObject = new objectName(parameter, parameter, etc..);
```

This will tell the objects **constructor** function to make an instance of the object, with the properties you pass as parameters. A constructor is just what it says – it constructs the new object for you. The constructor is almost always the function that carries the name of your object, as you will see below.

The Constructor

Since this is going to be a small little form object, we probably should simply call it by that name. So create an empty function with that name, and you will have your basic constructor:

```
function formObject()
{
}
```

It doesn't do anything yet of course. Calling this function isn't going to generate a lot of excitement. The next thing to do is decide which properties you want to set when you construct your first object of this type. We have our list above of properties, but we may not want to pass them all every time we make an object.

This will always be a unique design issue with your object and will depend on the situation. In our case, we probably don't want to have to specify colors each time, or the text size field. We'll save those for something else. But we do want to set the form name, so that it is unique with each object. So we'll make the form name the first parameter:

```
function formObject(formName)
{
}
```

Remember This?

We have already discussed that the **this** keyword always refers to the current object. In the case of the constructor function, that is what we want, we want to tell the object that is being created what **its** properties are supposed to be. We do this by saying:

```
this.property = some_value;
```

The **this** refers to the object being constructed, the property is the property you are identifying, and the value is what you are giving it. The property can be anything you want to call it (as long as it is not a JavaScript reserved keyword). Since we want to set a form name, we can call our property (and our parameter that we send for it) "formName" so now our constructor function looks like this:

```
function formObject(formName)
{
     this.formName = formName;
}
```

Keeping then names consistent helps you to keep things straight. So you have told the constructor: "When you build the object, set the formName property of my object to the value passed as the "formName" parameter." You set formName when you created an instance of the object:

```
var myObject = new formObject("coolForm");
```

So in this case, this new object you just created has a property called "formName" and its value is "coolForm."

Prototype properties

Okay, we can now create an instance of our **formObject** and pass it a form name. What about the colors and such? Well, we decided that those properties would be common to all objects in most cases, and that if they were to change they would change for all of them. What we need is some kind of "global" variable. We have this in the prototype property.
A prototype is exactly what the word means. Part of the definition of prototype is "An original or model after which anything is copied; the pattern.." So any prototype property you set will be a pattern for every instance of your object. They will all match (unless you override them later, which you can do.)

So for the rest of our properties, we'll set prototypes by naming the object, using the prototype keyword, making up a variable name, then giving it a default value:

```
formObject.prototype.backgroundColor = "#ffcc33";
```

It's as simple as that. Now every instance of our object has the property "backgroundColor = '#ffcc33'" attached to it. So our whole list would look like this:

```
formObject.prototype.backgroundColor = "#ffcc33";
formObject.prototype.textColor = "#003366";
formObject.prototype.textFieldSize = "20";
```

We can place this code just above our constructor, and now we have this code so far for our object:

```
formObject.prototype.backgroundColor = "#ffcc33";
formObject.prototype.textColor = "#003366";
formObject.prototype.textFieldSize = "20";

function formObject(formName)
{
     this.formID = formName;
}
```

Every time we build an instance of this little form object, it will carry the same three "global" or prototype values of backgroundColor, textColor and textFieldSize. Then each instance will have its own unique name that you pass it when you use the new keyword to build your object.

Now we need to actually write it to the page.

Making it happen

What we can do is write a function, which we will attach to our object as a method. This method will actually write our form object to the page. First, let's write the function that will be our method.

```
// methods for formObject
function formObject_writeControl()
{
}
```

It's empty right now, but we attach it to our constructor to let it know it now has a method. We can attach it to the constructor the same way you attach any other property:

```
function formObject(formName)
{
     this.formName = formName;
     this.write = formObject_writeControl;
}
```

Notice that you don't use the () of the function when attaching it to your object. Once again I simply made up a name for my object function, calling it "write". Now since this method is attached to our object, we can continue to use the **this** keyword to grab any property we want. This will allow us to set up the properties we are going to write out. We're going to use the **this** keyword twice this time, the first time (on the left side) to specify that it belongs to the current method running on the current object, and the second one (on the right side) referring to the current objects properties:

```
function formObject_writeControl()
{
     this.formName = this.formName;
     this.background = this.backgroundColor;
     this.text = this.textColor;
     this.field = this.textFieldSize;
}
```

Remember our prototype properties are available to any object, so we can freely use them here. Because the method is now part of the object, the method also knows the values of the prototypes (which are attached to the object!)

Now, using some document.write()s and inserting our variables in the right places, we can write the code that will build our object.

The code can get hairy here - particularly with the quotes. To avoid lots of escaping in JavaScript, which gets messy and hard to read, I will use single quotes for the inner values for our example. To build the statements is simply a matter of concatenating your method variables into the strings the **document.write()** is going to output. So "this.formName" puts whatever you passed as a parameter to the constructor as the form name in the actual form tag written to the page - just as if you had coded it by hand! You do the same thing for the text field and the submit button:

```
function formObject_writeControl()
{
     this.formName = this.formName;
     this.background = this.backgroundColor;
     this.text = this.textColor;
     this.field = this.textFieldSize;

document.write("<form id='" + this.ID + "'>")
document.write("<input type='text' id='someText' size='" + this.field + "'
style='background: " + this.background + "; color: " + this.text + "'
value='text' />");
document.write("<input type='submit' value='Enter' style='background: " +
this.background + "; color: " + this.text + "; font-weight: bold;' />");
document.write("</form>");
}
```

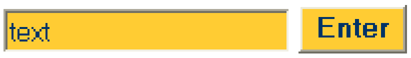
We make sure we close off the form as the last write.

The Moment of Truth

Now, our object should be ready to roll. Just one caveat: since we are using document.write() to output to the browser, you must write your method in the BODY of the page, not the HEAD. Your object code can go in the head, but when you actually call the method, it must be in the body. To write out your object, call the constructor and pass it a variable, and call the method:

```
<body>
<script type="text/javascript">
var anObjectForm = new formObject("myForm");
anObjectForm.write();
</script>
</body>
```

And viola! The code is written out for you with your parameter set. You should now see the exact same thing we hardcoded above:



Conclusion

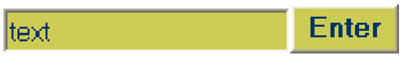
Try writing it out yourself and modifying to get the sense of how an object works. You can override a prototype (or even add one). To change the background color of one instance called yourObject, you could do this:

```
yourObject.prototype.background = "new_color";
```

And it would be different from the rest. So if I wanted the background olive green instead:

```
<body>
<script type="text/javascript">
var anObjectForm = new formObject("myForm");
formObject.prototype.backgroundColor = "#cc5";
anObjectForm.write();
</script>
</body>
```

I overwrote the prototype for the current image, and I get olive green:



Scripting The Select Object

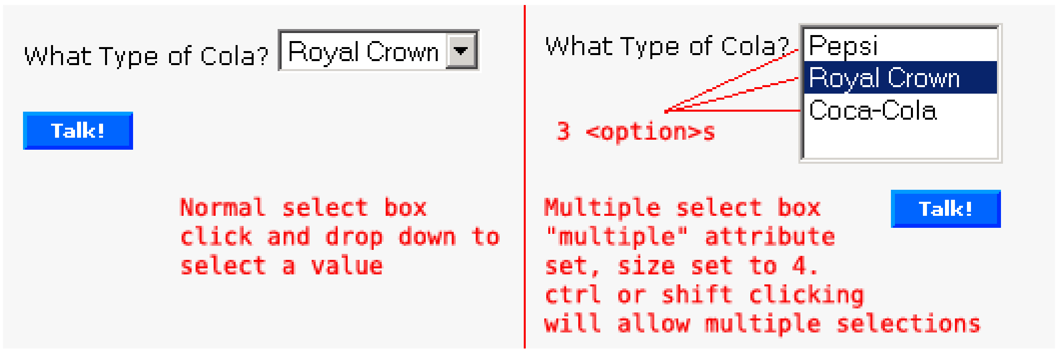
The Select object brings one word to mind: confusion. Beginning scripters generally look at this object and throw up their hands in despair. The select object can be very confusing because the select is made up of both the select object itself and its best friend, the option object. The two together create the drop down and multiple select lists of which we are so fond. This is generally where confusion sets in. When dealing with a select you are really dealing with two elements, not one. This can make references to the particular objects get long and confusing.

Luckily, it's not nearly as hard as it may seem. I'll explain how to get at the select with a script, and we'll write a useful script that we can add to the script library.

Taking a Closer Look at the Select

As you may have noticed by now, my standard practice is to examine the object first and understand it before we begin scripting. This is especially important with the select which is quite different from any other element.

The select object indicates that you are going to get a list of one or more values. Note, this list can be a drop down, or it can be a list in a box (scrollable or not, depending on the number of values and the size of the select). Below is an example of both:


What Type of Cola?
Royal Crown
Talk!
Normal select box click and drop down to select a value
What Type of Cola?
Pepsi
Royal Crown
Coca-Cola
3 <option>s
Multiple select box "multiple" attribute set, size set to 4. ctrl or shift clicking will allow multiple selections
Talk!

So you have a select object that holds nested within it one or more option elements that make up the list.

Let's have a look at the select object properties.

Table 1. Properties of the Select object

Property	Description
multiple	Indicates the user can select more than one option from the list (by shift or control clicking). Setting a select to multiple also gives you a list box instead of

	a drop down. That is, it is not a drop down list of single values, it's a list box showing two or more values. If you want a multiple select, you must also set the size of the box (how many options are shown at once) which is controlled by the "size" property, below.
size	Tells a multiple select the maximum number of options that can show in the list box. This does not limit the number of options that can be in the select, just the number that are displayed in the box. If there are more options than the size specified, the box will scroll the values.
type	Indicates the type of input element, in this case a "select" element.
selectedIndex	This indicates which option in the list is the current selection. The property returns an index value (a number) of its position in the options array (i.e, 0, 1, 2, etc.).
name	The "control name" just like any other form element.
length	This will tell you how many options are in your list. You can also manipulate this value to change your list.
options[index]	This references a specific option's properties within your select object. "Index" is the array number of the option you desire.

We also need to take a look at the option element. Even though the option element is an object in its own right, it can never be placed anywhere else other than within a select tag.

Table 2. Properties of the Option Object

Property	Description
value	The value of the option. In general terms, this matches the actual text that shows up in the list.
text	The actual text that shows up in the browser for the particular option.

A Quick Note About Netscape 4

```
I haven't talked much about browser compatibility or browser versions. In the
first case, since I am teaching you to use the DOM level 0  to access forms,
most of your scripts are going to work on most browsers. However, there are
some things that will choke Netscape 4. I have not had to code for NS4 in the
real world for more than a year, and chances are you won't have to either. NS4
makes up less than anywhere from 1-5% of the browser market *in general terms*.

However, it's always best to either check available logs or do some research
into your expected users to make sure that NS4 doesn't make up a part of your
users. The bottom line is I *may* call out to you when something will
definitely break NS4, but I won't go out of my way to code for that browser.
```

Talking to the Select

One thing that has really helped me learn is doing things one piece at a time, in small chunks. We can "talk" to the element by creating some code and then using JavaScript alerts to see if we are getting what we expect. We'll send the select some code asking it some questions, and we'll use the alert method to "talk" back and tell us if we are right.

This will help us understand the objects first, before we attempt to script them.

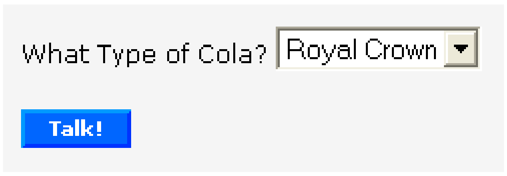
Let's start with this basic select list of delicious Colas (in order of taste, of course):

```
<form name="colas">
<select name="talkingSelect">
     <option value="cola1">Pepsi</option>
     <option value="cola2" selected="selected">Royal Crown</option>
     <option value="cola3">Coca-Cola</option>
</select>
<input type="button" onclick="TalkBack(this.form);" value="Talk!" />
</form>
```

A Quick Note About XHTML

```
You might notice in my HTML code that I do strange things like ending a tag
with /> or quoting minimized attributes like selected="selected". These are
merely rules for proper XHTML, which is a new recommendation by the W3C. If you
want to find out more about XHTML and why it is a good thing, visit the
following links:

http://www.w3schools.com/xhtml/ (W3 Schools)
http://www.w3.org/TR/xhtml1/ (W3C)
```

The little form looks like this:



This is a simple select with three kinds of cola as options. It's a single drop down list, the most basic type you see on a web page. The button next to the select is what we will use to fire off our test "talking" script. Notice, the parameter of the script will send the form object along to make referencing our select easier.

The Talking Script

Let's find out some things about the select object itself first. In doing so, you'll learn exactly how to get at the needed bits of a select.

Let's find out 4 things about this select:

The length: The number of options the list has
The selectedIndex: Which options is selected
The text of the selected option: What shows up in the browser for the current selection
The value of the selected option: The "value" attribute that is sent along during a form submission

The script will put each value into an alert and display a value, which if done correctly, will show us "3" for the length, "1" for the selectedIndex (Pop Quiz: Why not 2? Answer later), "Royal Crown" for the text and "cola2" for the value.

We'll do one at a time to keep it simple and clear. First, let's make a reference to our select object to make things easier. Remember, we have the form object sent as a parameter already. Once again using the elements[] array, we'll use the name to point to our select. This reference is placed in the theSelect variable.

```
var theSelect = theForm.elements['talkingSelect'];
```

We know from Table 1 that "length" is the property to tell us how many options are in the select, so we'll place that property's value in a variable called selectLength.

```
var selectLength = theSelect.length;
```

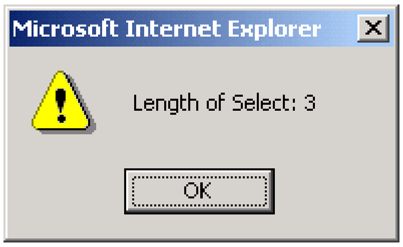
Now all we have to do is set up an alert (with some brief explanatory text) and place the selectLength variable inside to display. If you have never done this before, it is quite easy. The alert is a method of the Window object in JavaScript. It is what is called a "modal window" which means you must click the "OK" button before you can get back to the browser.

The syntax is simply alert(message). You can put the message inside your alert as a string ("remember, this is a string, it has opening and closing quotes around it") and it will display exactly as you write it. If you pass it a variable (without quoting it) you will see the variable's value. In our case we want both – we want some explanatory text of what the variable is so we don't have to remember them all. So we will concatenate (fancy word for join together) the two values using the + operator. You'll find that this technique is excellent for debugging and testing scripts.

Notice we put a space after the semicolon in our "Length of select: " string before we add the variable, otherwise they would butt up against each other and it would be harder to read.

```
function TalkBack(theForm)
{
     var theSelect = theForm.elements['talkingSelect'];
     var selectLength = theSelect.length;
     alert('Length of Select: ' + selectLength);
}
```

Okay, click the Talk! button, and you should see something like this:


Microsoft Internet Explorer

Length of Select: 3

This nicely matches up with the fact that we do indeed have three colas in our list. Being able to get the length of the select will be important later on when we are doing more advanced operations on the select.

Next, we'll ask the select which item is selected. We know based on our HTML code that Royal Crown is the one we hope to get:

```
<option value="cola2" selected="selected">Royal Crown</option>
```

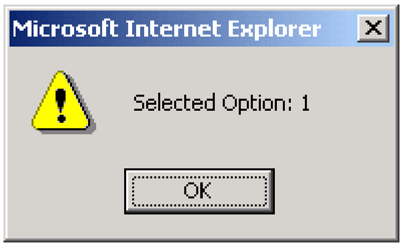
Based on Table 1, **selectedIndex** is the property we want, since that will return the index number of the option selected. Once again we attached that to our reference to the select and assign it to a variable like so:

```
var selectedOption = theSelect.selectedIndex;
```

And set up an alert to tell us when we press our button:

```
alert("Selected Option: " + selectedOption);
```

Add these to your script and run it. You should see:


Microsoft Internet Explorer

Selected Option: 1

The quiz question above asked why you wouldn't see 2 instead of 1. After all, it's the second item in the list, right? Answer: JavaScript array's start with 0, not 1. So the count of the cola array (the options array) is 0, 1, 2. The second option is 1.

So now we know that any time we want to know what option is selected in a select (a single select, mind you – see tip below), we simply find out what the **selectedIndex** property is.

```
Tip: If you have a multiple select (your select is set to multiple and has a
size value) and the user has selected more than one item, the selectedIndex
property only returns the top/first selection.
```

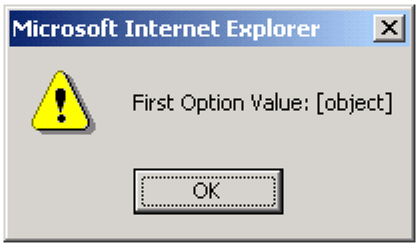
Hairy References

The next two properties are the troublemakers. This is where your references can get long and confusing. Luckily, it may look confusing but if you understand what is going on it makes perfect sense.

To review, remember that there is a property of the select called options[index], which allows you to access properties of a single option tag. So theoretically, if we wanted to get at the Pepsi option, we would do:

```
var myOption = theSelect.options[0];
```

And that would be correct. Once again we simply attach the property to the object reference. Now if you used an alert to show the value of myOption, you would see this:


Microsoft Internet Explorer

First Option Value: [object]

This is telling you that you indeed do have an object (the first option in your list) but since you didn't request any properties of that object, you didn't get any of that information.
This is where Table 2 comes in. Once we have reached our desired option, we can use the properties of the option object (not the select object) to get at the information we need. In our case we are looking for the display text. The property we want is, appropriately, "text." So we would do this:

```
var myOption = theSelect.options[0].text;
```

Using an alert to display that should give us the cola name we see in the browser. But wait, the specification we laid out says we want the cola name of the **selectedIndex**, the cola selected, not just the first item in the list.

Here's where people get confused. Remember, the options[] array simply wants an index number, in our case either 0, 1 or 2. Also remember, the **selectedIndex** returns – you guessed it – the index number of the selected item! Sounds like a match made in heaven. So, in the place of the 0 for the index, we want to put the **selectedIndex** number. Here's the gotcha – **it still has to be attached to the reference to your select object**. So it should look like this:

```
var selectedText = theSelect.options[theSelect.selectedIndex].text;
```

This confuses people because there are seemingly two references to the variable pointing back to our select. And this is true, there are. Why is this? Because the **selectedIndex** by itself has no meaning, so it would be undefined. JavaScript would wonder "**selectedIndex** of what?" The script is merely looking for an index value there, and the value for **selectedIndex** of itself is nothing at all. However, the value for theSelect.selectedIndex is in fact, 1. So the above line is exactly the same as:

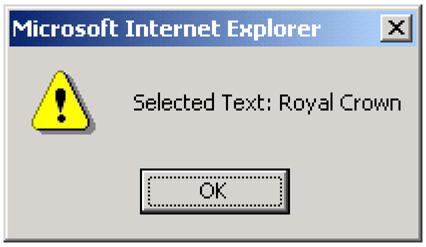
```
var selectedText = theSelect.options[1].text;
```

The reason we want to use **selectedIndex** and not just hard code the 1 in the line, is that unless we always want the same answer, using the 1 doesn't make any sense. What if the user changes the selection? Using **selectedIndex** ensures our script always works, regardless of which option is selected.

So, now we can add the following two lines to our script:

```
var selectedText = theSelect.options[theSelect.selectedIndex].text;
alert("Selected Text: " + selectedText);
```

Running your script should give you:



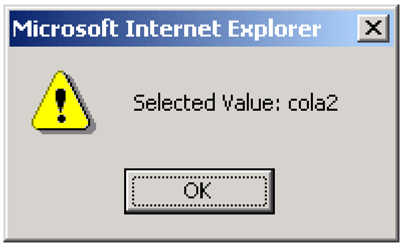
Microsoft Internet Explorer

Selected Text: Royal Crown

Now let's say we don't want the text, but the value of the option since we are doing some back end functionality work. We only want to know which cola was selected. The value property works the same as the text property. Simply add:

```
var selectedValue = theSelect.options[theSelect.selectedIndex].value;
alert("Selected Value: " + selectedValue);
```

Running your script should give you:


Microsoft Internet Explorer

Selected Value: cola2

So our finished script should look like:

```
function TalkBack(theForm)
{
     var theSelect = theForm.elements['talkingSelect'];
     var selectLength = theSelect.length;
     var selectedOption = theSelect.selectedIndex;
     var selectedText = theSelect.options[theSelect.selectedIndex].text;
     var selectedValue = theSelect.options[theSelect.selectedIndex].value;

     alert("Length of Select: " + selectLength);
     alert("Selected Option: " + selectedOption);
     alert("Selected Text: " + selectedText);
     alert("Selected Value: " + selectedValue);

}
```

Your Turn

Play around with this configuration, adding some options, moving the selected option around and getting different values. Get comfortable with exactly how you retrieve the various properties from the select and option objects.

Conclusion

The select object is really made up of two objects: the select and the option. You will never get confused on how to get the information you need if you simply reference the correct properties of the correct object.

And always remember, JavaScript arrays start at 0, not 1! Remember that the selectedIndex property cannot be used by itself, it must be attached to a reference to a select object.

Multiple selects behave the same way as single drop downs do, with the exception that getting the selectedIndex returns only the topmost item.

Scripting the Select: Moving Things Around

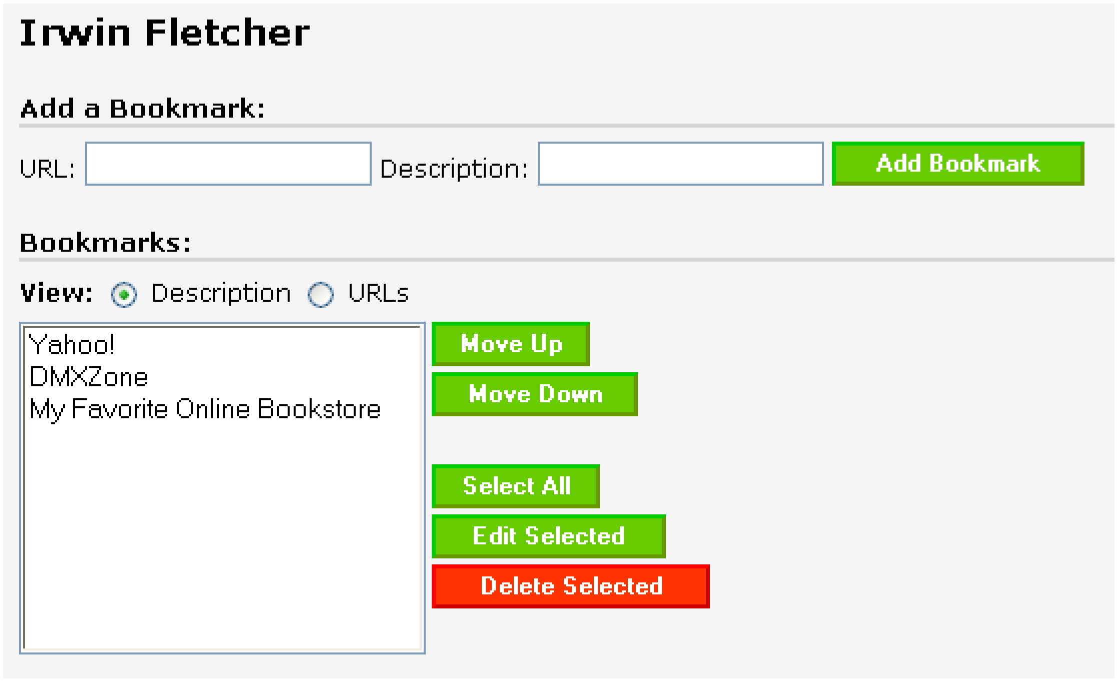
Often it isn't enough simply to be able to grab which OPTION the user has selected. What if you have a form where the user should be able to add or remove a value to the menu? Is it possible to actually change the values in the menu on-the-fly? What if we wanted to move a value up or down?

In fact, this type of functionality is often required by an application. Imagine a site where an administrator has to manage permissions. Maybe there is a SELECT list of sections the user has permission to see. The administrator needs to add or remove them (or initially set them up) as needed.

The Bookmark Application

For our script, we'll assume you are an administrator of a web site that allows a user to keep track of his or her own bookmarks. They will need to be able to add or remove bookmarks, as well as edit them. It would be nice if they could move them up and down (to change the order). Lastly, it would be really nice if they could allow them to view the list either by URL or by the description.

Remember, this wouldn't be an application all by itself, but a small part of a larger application. Let's take a look at what we are talking about:



The user comes to a page where their name is displayed at the top, and then their current list of bookmarks under the "Bookmarks" heading. At the top, the user can enter both the URL and the Description for a new bookmark, and add it by clicking the "Add Bookmark" button. You might have realized that we will have to handle what happens if the user forgets to input one of the values. We're assuming these bookmarks will be functional somewhere else in the application, so both the URL and the description need to be present.

Back in the Bookmarks section, the user can click buttons to move bookmarks up or down, or to delete or edit a selection. (Hmm, what if the user hasn't selected anything?) There is also a button to select all the entries should the user want to easily clear the list.

***Note**: Building an effective user interface that does so many things is challenging. What I have presented is sufficient for this paragraph. But in the real world, such a user interface would need to be tested on real users to make sure the design really works. Also realize that the end result of any actions taken in our little bookmark example would of course have to be implemented on the back end of the application. The choices would have to be saved in a database for retrieval each time the page is requested. So you could imagine the screen shot above would also probably include a save button or update button as well.*

Forms and Buttons

A very real consideration here is which, if any, button should be the submit button. It is good practice to only have one submit button for a form. If you have two submit buttons for a form that have different functions, you probably need to rethink your page.

In our case, the most likely candidate for the submit button would really be the unseen save button I talked about in the note above. The user would really only submit the page when they have completed manipulating their list.

Since we are really only manipulating the list anyway, we can probably make all of the buttons plain old buttons. Each button will call the particular function it needs to accomplish the desired task.

The Pieces of the Puzzle

As I mentioned above, I'm presenting this in bite size pieces. We will write the script to add a new option to the select list. Assume we already have the three entries as shown in the screen shot above in our bookmarks. Here is our SELECT:

```
<select name="bookmarkList" multiple="multiple" size="10">
     <option value="www.yahoo.com">Yahoo!</option>
     <option value="www.dmxzone.com">DMXZone</option>
     <option value="www.amazon.com">My Favorite Online Bookstore</option>
</select>
```

Notice this is going to be a multiple select, we are going to allow the user to select more than one item in the list if they wish. In addition, we want them to be able to see enough of the list to make it easy to manage, so we have set the size attribute to 10, so that 10 bookmarks can be viewed at once.

We also have the "add" portion of the form up top:

```
<h5>Add a Bookmark:</h5>
<label for="newURL">URL:</label> <input type="text" name="newURL" id="newURL"
/> <label for="newDescription">Description:</label> <input type="text"
name="newDescription" id="newDescription" /> <input type="button" name="add"
value="Add Bookmark" class="btn" />
```

We are again using the LABEL tag for usability and accessibility. Note the use of the H5 tag. Simply making that text bold and putting a break after it would have held little semantic meaning. Using the H5 tag gives us the line break we need, adds semantic value (it's a title, not just bold text) and we can handle the styling of it in a CSS style sheet. Do not neglect to include the name attribute for your buttons, this can often be a useful attribute in scripting. It isn't required, but it is good practice.

We have a radio group that will toggle which display the user desires to see:

```
<h5>Bookmarks:</h5>
<strong>View: </strong><input type="radio" name="view" value="viewDescription"
id="viewDesc" checked="checked" /> <label for="viewDesc">Description</label>
<input type="radio" name="view" value="viewURLs" id="viewURL" /> <label
for="viewURL">URLs</label>
```

The description is set as checked and the default view the user will see when entering the page.

Lastly are the control buttons for the list:

```
<input type="button" name="moveUp" value="Move Up" /><br />
<input type="button" name="moveDown" value="Move Down" /><br />
<input type="button" name="selectAll" value="Select All" /><br />
<input type="button" name="edit" value="Edit Selected" /><br />
<input type="button" name="delete" value="Delete Selected" />
```

Each button will trigger one of our scripts.

Adding a Bookmark

The first thing to tackle is adding a bookmark. This will entail adding an OPTION to the select. It's actually a fairly simple and straightforward process. We are going to need to process two pieces of information, the bookmark description and the URL, both of which come from the text fields at the top of our form. So let's pass those as parameters so the function can act on them:

```
function AddBookmark(url, description)
{

}
```

How do we pass those values to the function? We'll again make use of the **this** keyword and a correct form reference during the onclick event.

```
onclick="AddBookmark(this.form.elements['newURL'],
this.form.elements['newDescription']);"
```

This looks a little different than the form references I have talked about in the past. Normally, we would grab a form element like:

```
document.forms['formName'].elements['elementName'];
```

But once again the **this** keyword helps us out. If **this** refers to the current object (which is going to be the button which has the onclick), we can attach a form reference to that current object. That form reference (this.form) simply means "the form the current object is in." It's just another way of saying document.forms['formName']. After that, the rest is the same, the elements array with the element name.

One more thing has to be added. We don't just want a reference to the text field itself (which is what we would get with the above), we want the **value** (what the user entered.) Simply attach that property to the reference, and the function will get the information it needs:

```
onclick="AddBookmark(this.form.elements['newURL'].value,
this.form.elements['newDescription'].value);"
```

Now the script can act upon those parameters without doing any more work to them. One question might be asked though, why didn't I just set a variable inside the function and reference the elements there by name? Mmmm, sounds like time for a tip!

TIP: Abstract your functions

Had we referenced the elements inside the function, that function would only be usable on that one page. What if you need the same functionality on another page? You would have to duplicate the function in the other page. This is not only wasteful, if you need to make changes to the function, you'd have to find it on every page and change every instance. A maintenance nightmare.

However, if you write your functions so that they can be used by any page, you are working much smarter and saving yourself a lot of pain and frustration down the road. To abstract something means to divorce it from its local environment. That means you don't need to know how the function works, only how to use it. For the AddBookmark() function, all you really need to know is you must pass it a URL and description value. You could keep a function like this in a global file, and use it on every page, merely sending it the information it needs. Not every function should or even can be abstract, but it's a good idea to try and design them that way first. If you find it needs to be local to the page, then that is fine as long as you have explored your options.

The only other piece of information our script needs is where the values are going to be sent. We really should send along the SELECT object as well, so our function stays abstract. So we'll add a parameter for that and add it to the onclick call in the same manner as above.

```
function AddBookmark(url, description, oSelect)
{

}
onclick="AddBookmark(this.form.elements['newURL'].value,
this.form.elements['newDescription'].value,
this.form.elements['bookmarkList']);"
```

I like to use the naming convention "o" plus some descriptive name when dealing with form objects. It is somewhat of a convention (depending on who you talk to) as well. So I chose the name oSelect for our SELECT object.

Adding to the SELECT

To add an option to our SELECT we are going to use something called a **constructor**. A **constructor** is exactly what it sounds like: it constructs something. We won't get any deeper into them than that in this paragraph, since they are an advanced subject. But you don't need to know much about them to use them.

The **constructor** is basically a type of function, and is capitalized with the name of the object is builds. In this case, it builds an OPTION element, so it is of course called Option(). Adding the **new** keyword in front of it builds a new option for us.

To do it, we need to pass the constructor a couple of parameters. We have to tell it what is going to serve as the **value** property of the OPTION and what is going to serve as the **text** property (what the user sees). So in total, the format is:

```
new Option(text, value)
```

We only need attach that to our current options array. In order to do that, we must first get the **length** (how many) of the options array. **Length** is a method of the options array, and we can find out that value by assigning it to a variable like so:

```
optLength = oSelect.options.length;
```

We can't simply add the new option to oSelect.options – it must be added to the options array. Furthermore, we can't add it like so:

```
oSelect.options[] = etc.
```

because JavaScript sees that as a syntax error. Giving it the length as the value of the array will allow the script to add our new option at the end of the list, which is what we'd like to have happen anyway. Now we aren't going to send "value" and "text" as our parameters, we are going to send URL and description, which hold the values of the text that the user entered. So the line to add the option looks like this:

```
oSelect.options[optLength] = new Option(description, url);
```

That is the "action" line, the one that when executed really puts the new value in the SELECT. So the full script looks like:

```
function AddBookmark(url, description, oSelect)
{
     optLength = oSelect.options.length;
     oSelect.options[optLength] = new Option(description, url);
}
```

Give it a try. You should see the new option pop right into the select list. Now that works fine, but we have a few issues to deal with:

- What if the user enters an empty string (either value or text), should we allow it?
- What if the user enters a duplicate string?

Adding an empty string (to either value) would present problems, so let's not let the user do that. Also, there really shouldn't be any duplicates either. So before we finish, we should check for both of those conditions, and not allow an add if either rule is violated. Of course, to make the site user-friendly and more usable, if we don't allow a value, we need to let the user know **why**.

We'll check for the empty strings first. However we need to make the distinction between "no data" and "bad data." No data means the user has not entered anything into the text field. However, a simple space will constitute a filled in text field. However, one space does not a valid URL make, as Confucius used to say

We could use **regular expressions** to check for bad data, and tell the user if it does not match our criteria. Regular expressions are quite an advanced topic however, so I won't cover it, at least for now. We'll assume for our purposes that if the user has input data that it is good data (a horrible assumption in the real world!) If you want to read up on your own, you can take a look at http://wsabstract.com/javatutors/redev.shtml.

So, checking for an empty text box is simple. We can use the **not** operator, which is the exclamation sign (!) on our parameters.

```
if(!url)
```

The beginning of this condition says "if not URL." In other words, if URL has no value. That's exactly the test we want to make. If this evaluates to true, we should set up an alert to tell the user the problem, then get out of the function so the add is not executed. But how do we get out of the function? We use a **return**. A return simply says, begone!, go back!, return to whence you came from! (as Shakespeare used to say). In other words, you exit the function right at the return statement. Since we will exit the function **inside** the if condition, nothing else will be executed, and the blank data will not be added to our list.

```
if(!url)
{
     alert("Please enter the URL for your bookmark.")
     return;
}
```

You handle the description exactly the same:

```
if(!description)
{
     alert("Please enter the description for your bookmark.")
     return;
}
```

The last thing we need to check for is a duplicate value. What needs to be done is check both the URL against the values of the current OPTION and the description against the text of the current OPTION.

This is a job where the **for** loop comes in handy. We have used this previously when iterating through radio button groups. What we want to do is loop through each option, and check to make sure none of the values are duplicates. Now, we've already specified the length of the options array in our script, which we need to use in the **for** loop. We can just move that up to the top of our function so it will be available for our loop.

```
for(i=0; i<optLength; i++)
{
}
```

Next, set a variable for both the value and the text of the current option. We can get that information by accessing the value and text properties of the options array.

```
optionText = oSelect.options[i].text;
valueText = oSelect.options[i].value;
```

Remember, we already specified oSelect as our SELECT object. We're simply taking the current option in the array, which is indexed by the variable "I" set in our **for** loop. So it's the same as oSelect.options[0].text, then oSelect.options[1].text and so on and so on. Now we need to check the values the user inputted (represented as our parameters) against these. We'll use a nifty little method JavaScript has called **toLowerCase()** to help us. This will simply put both strings we are comparing (the value or text of the current option versus the value or text of the users input) in lower case. So the comparison is a bit simpler, and it will also catch duplicates that have different capitalizations. Note that this doesn't actually change what the user typed it; in keeps his/ her capitalization – it's just so that we're comparing apples with apples rather than oranges with Oranges.

```
if(optionText.toLowerCase() == description.toLowerCase())
```

If this evaluates to true, we tell the user the problem and we return out of the function so the duplicate data is not entered.

```
if(optionText.toLowerCase() == description.toLowerCase())
{
     alert("That bookmark already exists.")
     return;
}
```

Do the same for the URL as well.

```
if(valueText.toLowerCase() == url.toLowerCase())
{
     alert("That bookmark already exists.")
     return;
}
```

You could do this all in one condition using the logical OR operator (||), but I wanted to separate it for clarity. Now the full function looks like this:

```
function AddBookmark(url, description, oSelect)
{
     var optLength = oSelect.options.length;

     if(!url)
     {
           alert("Please enter the URL for your bookmark.")
           return;
     }
     if(!description)
     {
           alert("Please enter the description for your bookmark.")
           return;
     }

     // check for duplicate values
     for(i=0; i<optLength; i++)
     {
           optionText = oSelect.options[i].text;
           valueText = oSelect.options[i].value;
           if(optionText.toLowerCase() == description.toLowerCase())
           {
                 alert("That bookmark already exists.")
                 return;
           }
           if(valueText.toLowerCase() == url.toLowerCase())
           {
                 alert("That bookmark already exists.")
```

```
                    return;
              }
        }

        oSelect.options[optLength] = new Option(description, url);
}
```

Conclusion

Play around with the script, as usual to see what you can do with it. See if you can combine the two if statements with a logical or to check for all duplicate values at once. Either way would be correct, although you could cut down on the code a bit.

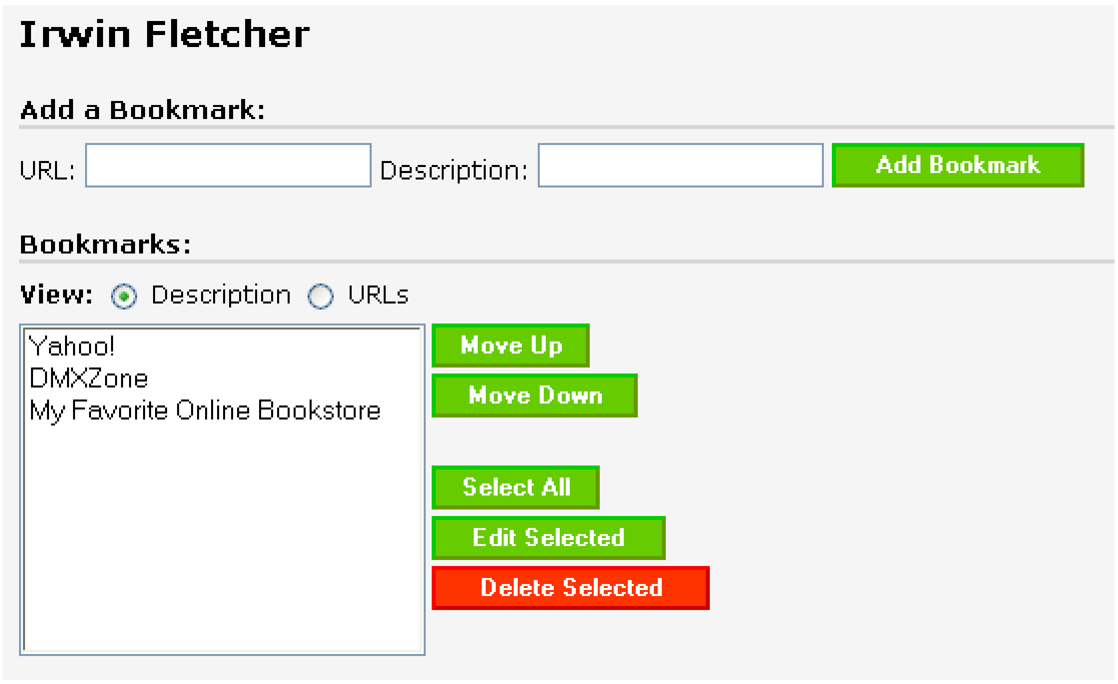
The Option() constructor actually "constructs" the new option for you, there is no need to do any complex scripting to add an option to your list. Furthermore, since we kept all the page specific script information outside the function, it is abstract enough to be used anywhere in our application.

Scripting the Select; Going further

I'll continue to extend the functionality to make the tool more useful. In the process, you'll learn how to further manipulate the OPTION array within the SELECT object.

A Quick Review

Once again, the bookmark editor looks like:



Irwin Fletcher

Add a Bookmark:

URL: Description:

Bookmarks:

View: [✓ Description] URLs

Yahoo!
DMXZone
My Favorite Online Bookstore

The HTML for the editor:

```
<form name="bookmarks">
<h3>Irwin Fletcher</h3>
<h5>Add a Bookmark:</h5>
<label for="newURL">URL:</label>
<input type="text" name="newURL" id="newURL" />
<label for="newDescription">Description:</label>
<input type="text" name="newDescription" />

<input type="button" name="add" value="Add Bookmark"
onclick="AddBookmark(this.form.elements['newURL'].value,
this.form.elements['newDescription'].value,
this.form.elements['bookmarkList']);" />

<h5>Bookmarks:</h5>
<strong>View: </strong>
<input type="radio" name="view" value="viewDescription" id="viewDesc"
checked="checked" />
<label for="viewDesc">Description</label>
<input type="radio" name="view" value="viewURLs" id="viewURL" />
<label for="viewURL">URLs</label>

<select name="bookmarkList" multiple="multiple" size="10" style="float: left;">
     <option value="www.yahoo.com">Yahoo!</option>
     <option value="www.dmxzone.com">DMXZone</option>
     <option value="www.amazon.com">My Favorite Online Bookstore</option>
</select>

<input type="button" name="moveUp" value="Move Up" /><br />
<input type="button" name="moveDown" value="Move Down" /><br />
<input type="button" name="selectAll" value="Select All" /><br />
<input type="button" name="edit" value="Edit Selected" /><br />
<input type="button" name="delete" value="Delete Selected" />

</form>
```

Notice that we used the **LABEL** tag as described previously for usability and accessibility. You could also use the **TABINDEX** attribute if you desire.

Why TABINDEX?

When a web page is loaded, a user can immediately use the tab key to begin tabbing through links and form elements. In general terms, this works well when a page is cleanly authored and built in a logical order. But sometimes you end up with a page that presents a default tab order that doesn't make sense at all. This makes it difficult for a user who relies on the keyboard instead of the mouse to navigate a page. (For example, a person with some kind of repetitive stress injury, or maybe just a broken writing arm). You also may want a different order that the default order, because it makes more sense for your site.

In such cases, you can add the TABINDEX attribute to both <a> and <input> tags. The TABINDEX accepts a numerical value, beginning with 1 (for the first item tabbed to), and then continues on until all the links and or inputs have values. The browser then presents the tab order as you have described it, instead of by default. When well done, it becomes simple to easily tab through links and form elements on a page. An accessibility feature as well as a usability feature!

If you accidentally give two elements the same index, the one that comes first in the HTML will be tabbed to first. You also don't have to explicitly tell every link and input on the page what its TABINDEX should be. The browser will rank the explicitly indexed links before ones that have no rank, again according to their appearance in the HTML. It's a good feature to use with complicated features, and many users will thank you for their use.

```
<script language="javascript" type="text/javascript">
function AddBookmark(url, description, oSelect)
{
     var optLength = oSelect.options.length;
     if(!url)
     {
           alert("Please enter the URL for your bookmark.")
           return;
     }
     if(!description)
     {
           alert("Please enter the description for your bookmark.")
           return;
     }

     // check for duplicate values
     for(i=0; i<optLength; i++)
     {
           optionText = oSelect.options[i].text;
           valueText = oSelect.options[i].value;
           if(optionText.toLowerCase() == description.toLowerCase())
           {
                 alert("That bookmark already exists.")
                 return;
           }
           if(valueText.toLowerCase() == url.toLowerCase())
           {
                 alert("That bookmark already exists.")
                 return;
           }
     }
     oSelect.options[optLength] = new Option(description, url);
}
</script>
```

Moving Options

We'll try and **abstract** our script as much as we can while we write it. Remember, this allows the script to be reused with little or no modification, as well as allowing somebody else to use the script without having to know the inner workings of the script itself.

One good way to start a script is to think about what it will need. When dealing with forms, I always want a reference to the form. Secondly, having a reference to the current object (in this case, our **SELECT** object) is always a must-have. Lastly, it would help if we could send the function which direction the user wants to move the bookmark (up or down). Having the direction will allow us to use one script for both the up and down movement.

Let's start off our script with those three things as expected parameters:

```
function MoveOption(oForm, upOrDown, oSelect)
{
}
```

As usual we use a nice descriptive name for the function, capitalizing the first word to distinguish it from a variable name. Note the "o" before the word Form and Select. I noted that often it's helpful to name variables that will refer to objects in this manner (think of it as short for "objectForm" or "objectSelect"). In the middle we have upOrDown, which will tell us which way to move the OPTION.

This function is going to be called from both the "Move Up" and "Move Down" buttons, using the **onclick** event. In the case of the up button, the event call will look like:

```
onclick="MoveOption(document.forms['bookmarks'], 'up',
document.forms['bookmarks'].elements['bookmarkList']);"
```

Notice the three parameters we are sending:

- The form name ("bookmarks") for the parameter "oForm"
- "up" for the parameter "upOrDown"
- The select name ("bookmarkList") for the parameter "oSelect"

The script now has all the basic information it needs to function correctly. Anything else we might need can be derived from these three parameters.

The way this is accomplished is simple. We'll take the VALUE and TEXT values of the current selection, and swap those with either the one above (if the user moves the option up) or the one below (if the user moves the option down.) Knowing this, there are 4 more local variables we can set up to make things easier:

- One for the selectedIndex (the option the user chose)
- One for the new position where the current option will move
- One to hold the new VALUE and TEXT values
- And one to hold the old VALUE and TEXT values

We'll set them up like so:

```
var choice = oSelect.selectedIndex;  // the current selection
var newPosition;                     // where it will go
var oldVal = oSelect[choice].value;  // old VALUE and TEXT values
var oldText = oSelect[choice].text;  // new VALUE and TEXT values
```

Notice how we used our "oSelect" parameter to derive three of these variables. The first one should be well known to you now, **selectedIndex** as the current selection. The second variable we merely declared for later use, it has no value yet. For the last two, notice how I used the first variable ("choice") to shorten the reference. Don't get discouraged if these references get confusing! It takes awhile to get used to them, and sometimes a so-called seasoned veteran like me has to take a step back and scratch my head.

Error Check

We have to be careful here. The best scripts take into account not only what Is expected, but what is unexpected. What if the user were to click one of the move buttons without having selected an option? Our script would blow up pretty quick. In fact, you really need to find this out **before** you attempt to set the third variable ("oldVal") since that declaration uses the **selectedIndex** property.

Verifying a selection is quite simple. All you need to do is make sure that the **selectedIndex** property is not equal to -1. If a SELECT has a value of -1, it is telling you nothing has been selected. So we'll make that check and send the user an alert if there is no selection. If the user has not done his job, we'll use the return statement to exit out of the function to allow them to make a selection.

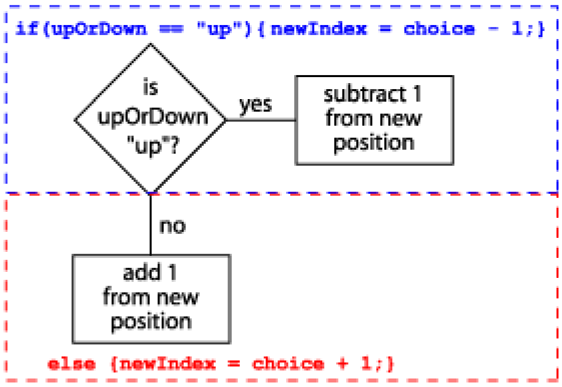
```
if (choice == -1)
{
     alert("You must first select the option to reorder.");
     return;
}
```

Since we already had the variable "choice" set as the selectedIndex, we can use that to make the comparison. Notice the double "=" for the comparison. Nothing trips up beginner (and expert!) scripters alike like forgetting the second equal sign when comparing values.

Where Is It Going?

Now we're rolling! We've made it through the error check and it's time to find out which way the OPTION is going. This calls for an IF…ELSE control structure. You've seen the IF control, the ELSE will simply give you a second option. In plain English, you would say it like this:

"If the user wants to go up, then subtract 1 from the new position variable (subtract, because you are going up in the list, like place 3 to place 2), otherwise (or..ELSE) add 1 to the new position variable (since you are going down in the list, like place 4 to place 5)."


if(upOrDown == "up"){newIndex = choice - 1;}
is upOrDown "up"?
yes
subtract 1 from new position
no
add 1 from new position
else {newIndex = choice + 1;}

Programmatically it looks like:

```
if(upOrDown == "up")
{
newIndex = choice - 1;
}
else
{
newIndex = choice + 1;
}
```

It's the classic fork in the road, give me the shortcut to Grandma's house or the long way around to the cottage. If, or else. If the upOrDown value isn't "up" it must be "down." Now if you are anal... I mean... smart... you are wondering "what happens if I accidentally misspell (or place any other value than "up" or "down") in my onclick call?" Certainly, this could happen, especially on a large application where you aren't the only one using your function.

The answer is that if you are concerned about this happening, you would add another IF statement after the ELSE, and then handle any other situation with some kind of error response, for example:

```
if(upOrDown == "up")
{
newIndex = choice - 1;
}
else if(upOrDown == "down")
{
newIndex = choice + 1;
}
else if((upOrDown != "up") && (upOrDown != "down"))
{
     alert("The programmer goofed!");
     return;
}
```

The last condition could check and see if neither value was present, and exit the function. There are much better ways to do such a check but you get the idea without me throwing too many things at you all at once. Generally, you shouldn't have to make such a check, especially if you are authoring your own pages.

Now we know which way the user wants to go. Now we've got to make something really happen.

Doing the Move

You might be wondering why we just don't set the newPosition variable (with the added or subtracted value) to the **selectedIndex** current variable and be done. In fact, you could do that. However, what you would get is the **selectedIndex** itself moving up or down, but not the values! Not quite the intended result. Remember, we have to swap the VALUE and TEXT values of the OPTION first, so the OPTION actually "moves."

In order to do the swapping, start out by setting the *newPosition's* VALUE and TEXT to the *user's* original selection. This will have the effect of taking the values above (or below) the user's choice and copying them there. If you were to stop the function there, you would have to concurrent duplicate OPTIONs. Here's how it is done:

```
oSelect[choice].value = oSelect[newPosition].value;
oSelect[choice].text = oSelect[newPosition].text;
```

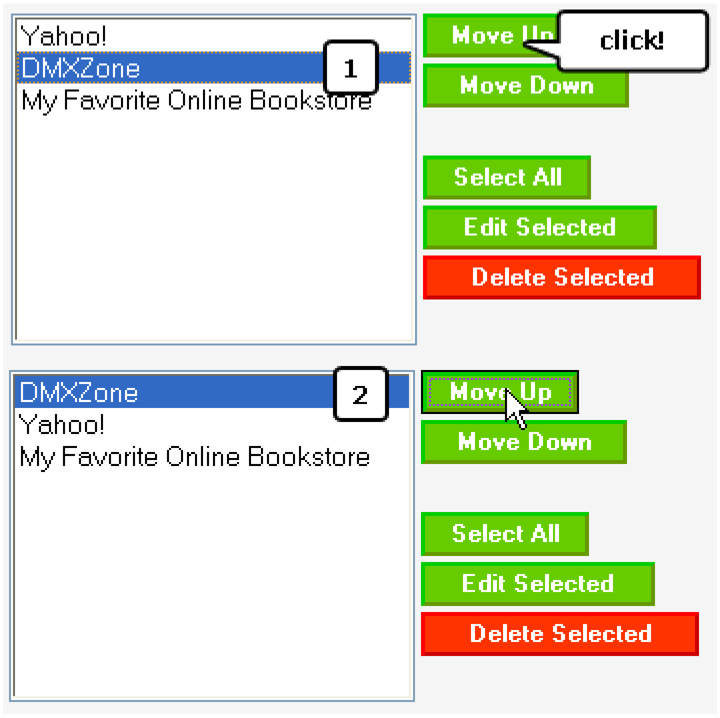
Remember the variable "choice" is the user's **selectedIndex**. The newPosition variable is that **selectedIndex** plus or minus 1. Now, you would do exactly the opposite, make the newPosition values take on the user's original selected values:

```
oSelect[newPosition].value = oldVal;
oSelect[newPosition].text = oldText;
```

Now at this point, you can finally set the selectedIndex to the newPosition, and finalize the move:

```
oSelect.selectedIndex = newPosition;
```

Below is a screen shot of what you should see if you moved an item up:



Gotcha!

Ah, there's always a gotcha, isn't there? A "gotcha" is something you might not expect to happen, but will break your <insert task here>. In this case, once the DMXzone bookmark has been moved to the top, a click on the Move Up button again will throw an error. Why? It's not because you can't move it again – you can. It's because you are trying to copy values that exist *above* the bookmark. And now you see the problem – there are no values above the top bookmark!

The same thing will happen if you attempt to move the bottom OPTION down. There are two ways you can handle this. First, you could check and see if the top item is first, or the last item is last. If either case is true, you can simply alert the user they can't go up or down further.

Or, you could have the item wrap, the top goes to the bottom, the bottom goes to the top. It's certainly more interesting for a tutorial, but Bruce (your friendly editor) informs me that were we to implement such a feature on a real site, usability experts would break your door down at 6am and drag you off to Camp Delta. So always be aware of the usability issues of your scripts. (Personally, I hate getting up at 6am.) Since it's the copying of the non-existent values that throws the error, we need to make our check before that happens.

If you find the user is at the top or the bottom, all you need to is one of two things:

If they are already at the top, set the newPosition variable to the last item (instead of the non-existent one above the top item). You can do this by subtracting 1 from the length of the options array.

If they are already at the bottom, set the newPosition variable to the top item (instead of the non-existent one below the last item). This is done by simply setting the new position to 0. Remember, 0 is always first in JavaScript arrays. View the code below:

```
//make it last if its already at the top
if(newPosition < 0) newPosition = oSelect.length - 1;

// make it first if its already at the end
if(newPosition >= oSelect.length) newPosition = 0;
```

Notice how I kept the whole IF statement on one line. When you have a short IF control structure, it is allowable (and sometimes nice to save space) to keep in on one line and avoid the bracket set. These two conditions will make sure that you have something to copy from and therefore, the script avoids any errors.

Here is the script in full, with comments:

```
function MoveOption(oForm, upOrDown, oSelect)
{
     // The user's current selection
     var choice = oSelect.selectedIndex;
     // Where the user wants the option to go
     var newPosition;
     // Save the user's VALUE and TEXT values so they
     // can be copied to the new location.
     var oldVal = oSelect[choice].value;
     var oldText = oSelect[choice].text;

     // If the user has not made a choice, exit the function
     if (choice == -1)
     {
           alert("You must first select the item to reorder.");
           return false;
     }

     // Find out which way the user wants the option to go
     // Then change the the "choice" variable (the current selection)
     // accordingly.
     if(upOrDown == "up") newPosition = choice - 1;
     if(upOrDown == "down") newPosition = choice + 1;

     // Check to make sure there is an option to copy.
     // If at top, wrap to the bottom and vice versa.

     // make it last if its already at the top
     if(newPosition < 0) newPosition = oSelect.length - 1;

     // make it first if its already at the end
     if(newPosition >= oSelect.length) newPosition = 0;
```

```
      // Copy the new VALUE and TEXT values (above or below) to the
      // user's current location
      oSelect[choice].value = oSelect[newPosition].value;
      oSelect[choice].text = oSelect[newPosition].text;

      // Copy the user's VALUE and TEXT values to the
      // new location, finalizing the swap of values
      oSelect[newPosition].value = oldVal;
      oSelect[newPosition].text = oldText;

      // Move the selectedIndex to where the user
      // "moved" the OPTION
      oSelect.selectedIndex = newPosition;
}
```

Conclusion

At this point you may have realized just what went on – you didn't "move" anything! What really happened is you simply swapped values. The only thing that really moved was the user's selection, and that only to match where his values were copied.

Notice how this script should be able to be used by anyone. It is "abstract." You should be able to tell a fellow developer to call the function via an onclick event, sending it the form, the direction to move, and the select object, and the move will be accomplished. They need know *nothing* about how the function actually works. This makes it easier to use, and easier to maintain.

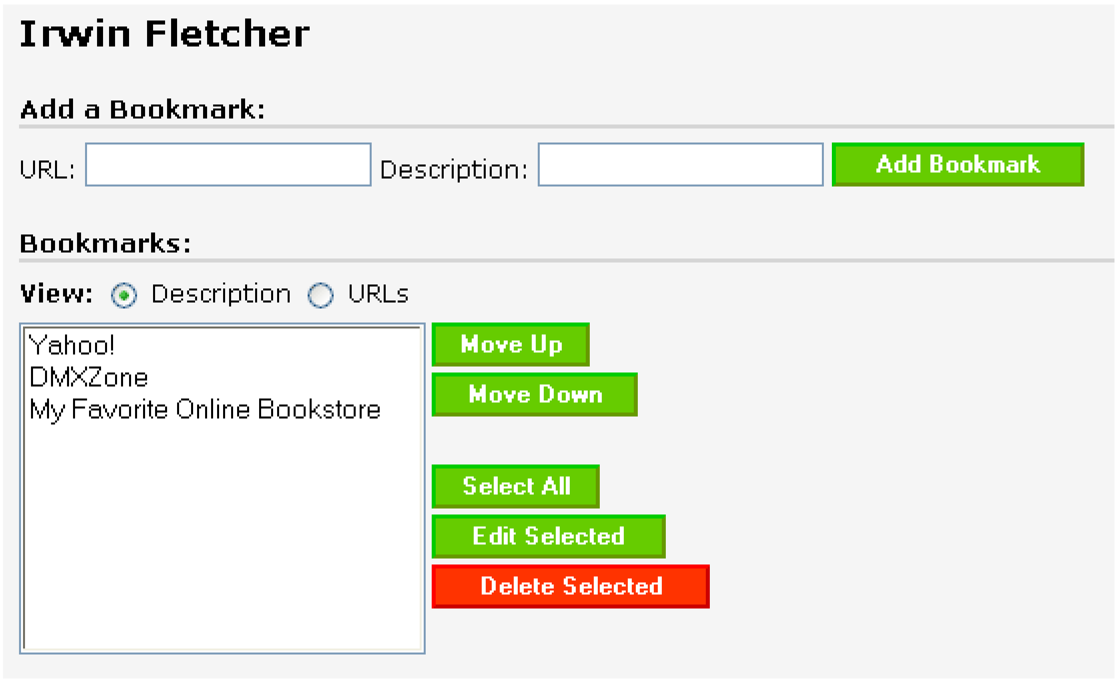
As I've stated before, the SELECT object and its references can be tricky to grasp. Tear apart the HTML and the script and try different things out. This is often the best way to understand a script – tear it apart!

Scripting the Select: Finishing Up

In the previous paragraphs I showed the user can add new bookmarks (and thus, new options) to the original SELECT and how the user could move an option up or down. The last things we need to accomplish are delete a bookmark, and select all the bookmarks. Selecting all the bookmarks is particularly necessary not only as a user feature, but for the form to function correctly as we will see later.

A Quick Review

Once again, the bookmark editor looks like:



The HTML for the editor:

```
<form name="bookmarks">
<h3>Irwin Fletcher</h3>
<h5>Add a Bookmark:</h5>
<label for="newURL">URL:</label>
<input type="text" name="newURL" id="newURL" />
<label for="newDescription">Description:</label>
<input type="text" name="newDescription" />

<input type="button" name="add" value="Add Bookmark"
onclick="AddBookmark(this.form.elements['newURL'].value,
this.form.elements['newDescription'].value,
this.form.elements['bookmarkList']);" />

<h5>Bookmarks:</h5>
<strong>View: </strong>
<input type="radio" name="view" value="viewDescription" id="viewDesc"
checked="checked" />
<label for="viewDesc">Description</label>
<input type="radio" name="view" value="viewURLs" id="viewURL" />
<label for="viewURL">URLs</label>

<select name="bookmarkList" multiple="multiple" size="10" style="float: left;">
     <option value="www.yahoo.com">Yahoo!</option>
     <option value="www.dmxzone.com">DMXZone</option>
     <option value="www.amazon.com">My Favorite Online Bookstore</option>
</select>

<input type="button" name="moveUp" value="Move Up" /><br />
<input type="button" name="moveDown" value="Move Down" /><br />
<input type="button" name="selectAll" value="Select All" /><br />
<input type="button" name="edit" value="Edit Selected" /><br />
<input type="button" name="delete" value="Delete Selected" />

</form>
```

Deleting Options

We must provide the user a way to delete a bookmark in case they have accidentally entered one they didn't want to, or one has gone out of date – also, there's few things that make bookmarks useless than so many book-marks you can't find what you're looking for, so the ability to delete is vital in a real-world situation.

Building the Delete Script

As usual, anytime we want to make a change to something in the OPTION list, it must be selected first. So we know that we must make a check and make sure a selection is made. As is typical with our other functions, we want to send along the FORM object and the SELECT object to make referencing things easier.

We're going to allow the user to delete as many options as they wish. Therefore we'll need to check each option to see if it is selected, and if it is, delete it. This is a particularly nasty gotcha with this type of operation on a SELECT, and we'll talk about that in a moment.

So we have the beginnings of our function as usual:

```
function DeleteOption(oForm, oSelect)
{

}
```

There isn't going to be a whole lot more to do. We simply need to iterate (technical term for "loop through" or "step through") the options array, check for a selection, and delete the item if its selected. So as in the past, we'll set up a variable for the length (how many there are) of our options array using the length property of the SELECT object, which returns the number of options in the options array:

```
optLength = oSelect.length;
```

Gotcha Time

Here comes the tricky part. We need a FOR loop here, so we can iterate through the options. You've normally seen it like the following:

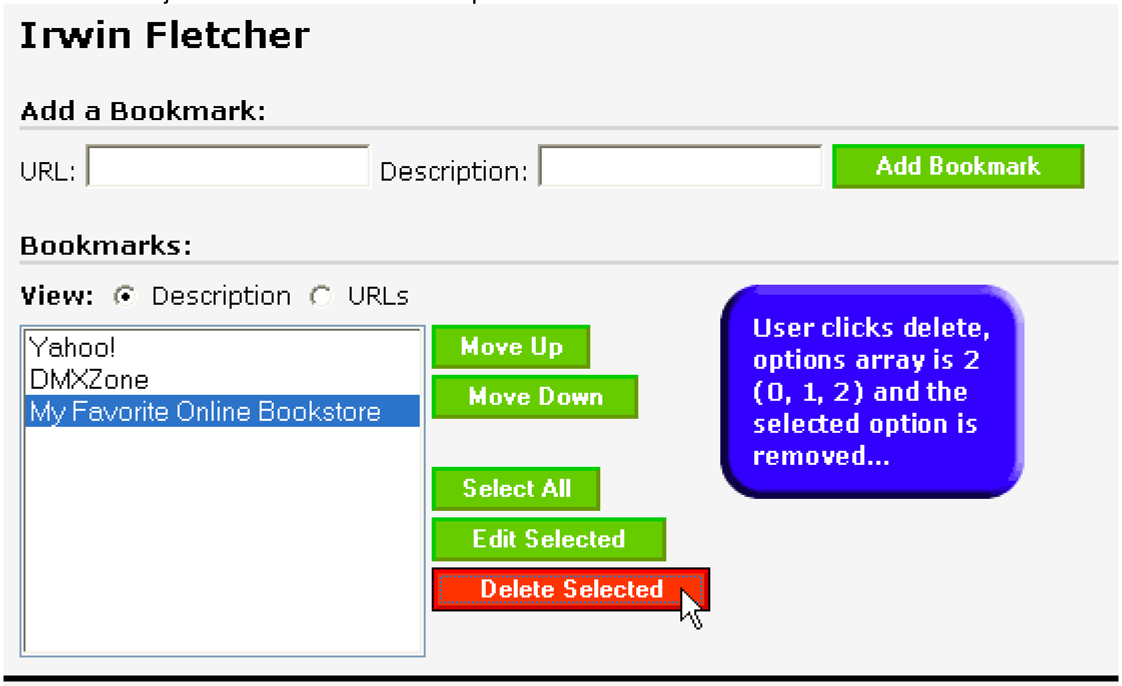
```
for(i=0; i>someVariable; i++)
```

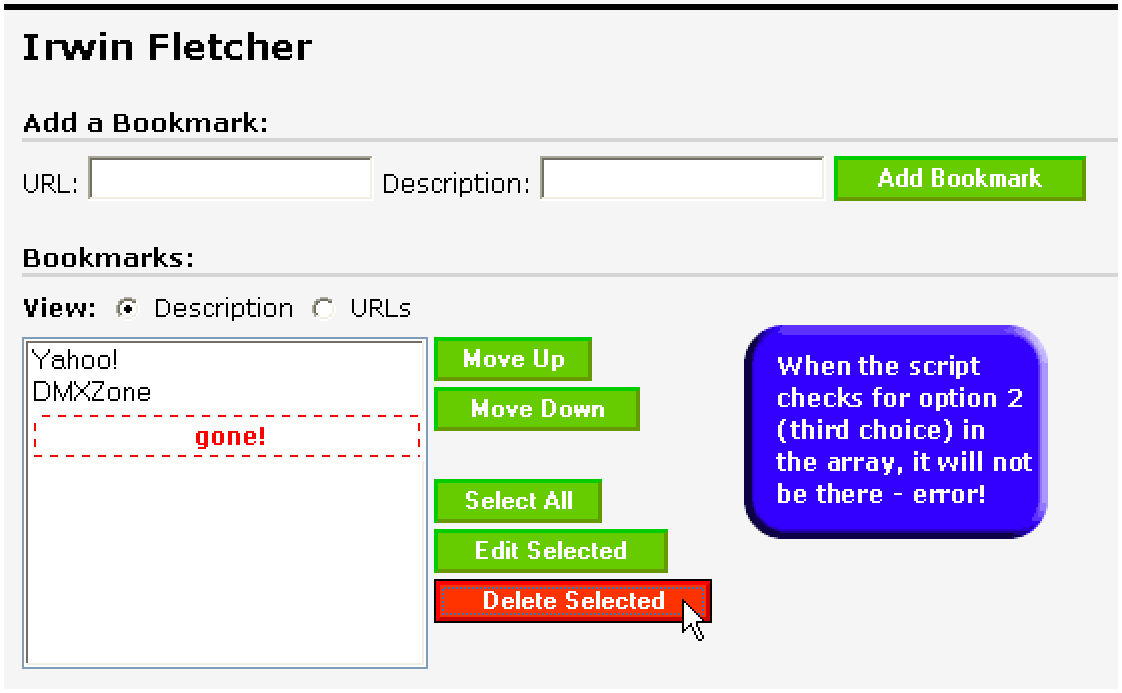
Again, the three parameters in the **for** loop are: 1) the variable declaration you are using to iterate with, and its value (usually you start at 1 or 0), 2) the condition to keep iterating (generally matching the value of i to some variable condition) and 3) what to do when the second parameter is true (usually increase the iteration by one.)

The problem is, if we do that it will **break our script**! Why? Remember that we are deleting options. At the beginning of our script we set a variable that has the length of our options array. What makes this even more of a gotcha is that doing it this way will work **sometimes**, but not all the time. This is the worst kind of error because it may seem part of the time, or even most of the time that the script works perfectly. In such cases it is difficult to know where to begin looking for the probem. Even worse, maybe the problem isn't even caught in the testing phase and is only discovered live, when a user triggers the error. So if we can avoid such an error in the first place, all the better.

Here is why the script breaks. In our example, the initial value for **optLength** is going to be 2. (Remember, it's not 3 because JavaScript arrays start at 0). At some point in your script, you are going to attempt to check and see if a particular option in your array is selected – **but there is not going to be an option there** – it has been deleted. We've set the optLength variable to 2 (for 3 items) but we have deleted at least one. Even worse, *depending on where that one was*, your script may or may not function correctly.

This may sound a bit confusing, but the key takeaway here is that trying to iterate **up** through the options array is going to be a problem, because sooner or later the script is going to try and check for an option that does not exist. This is because the length of the options array has changed, but we have not adjusted for that in the script.





The untidy result is when the user attempts to remove what is not present, a JavaScript error occurs. If the user is allowing errors to be shown (based on their browsing preferences) they will see a JavaScript alert, and of course the delete function will have not worked. Worse, if they are not allowing errors to be shown, the script will simply not work. They may or may not see some kind of indicator that there is an error, and simply assume the site is not working. This is the type of thing that gives web developers nightmares! We want to avoid such situations.

What's the solution? There are two things. First, we must adjust for the changing options array length. This means each time through the iteration, we must subtract one from our options array length. This will reflect the true length of the array after an option has been removed.

Secondly, we need to iterate DOWN through the loop, not up. Let me illustrate this since it is probably becoming confusing:

```
for(i=optLength-1; i>=0; i--)
```

You see, this is basically the reverse of the usual FOR loop. Instead of setting a initial value for i as 0, and counting up to a number, we're setting i as the total number of options, and counting backward to 0 (which will be the first element of the options array).

Why `optLength – 1`? The length property of the SELECT is going to give us 3 – which is incorrect for the array. Since JavaScript arrays start at 0 – we need 2. (0, 1, 2). (You know that it's traditional for me to point that out in every tutorial!) Therefore, we subtracted one to give us the correct value. (Inconsistent? Maybe. There is probably a good reason for it which is outside my sphere of knowledge. Just file it under "things to remember!")

Then the second parameter, our "continue as long as" condition checks to see if i is greater or equal to 0. Since 0 is the first item in the array, when i becomes -1 we stop – since we've gone beyond the first item and need not check anymore. Lastly our **`i--`** decreases instead of increases i so we're moving down, not up.

Now, our script will never meet a condition where there is no option. We only delete one option at a time (even if the user is deleting all of them). Since we are going down the length of the options array, and are checking the HIGHEST number first (in our case, 2) we'll never check one that isn't there.

The Actual Deleting

Actually deleting the option is simple. Merely set the current option in the array to "null" (null meaning "nothing"):

```
oSelect.options[i] = null;
```

Where [i] is the array position (remember, we are in a FOR loop). However, we don't want to do that to ALL the options – we might delete something that was not selected. Therefore, first we need to check and see if the current option is selected, then delete it if it is.

This is also very simple to do. SELECTED is a property of the OPTION element. If that property evaluates to true, then the option is selected. So we can set up an IF condition to check and see if the current option is selected, if it is, we delete it:

```
if(oSelect.options[i].selected == true)
{
oSelect.options[i] = null;
}
```

As you see, only options that are selected will be deleted – whether it is one option, or all of them.

The whole function looks like this:

```
function DeleteOption(oForm, oSelect)
{
     optLength = oSelect.length;
     for(i=optLength-1; i>=0; i--)
     {
           if(oSelect.options[i].selected == true)
           {
                 oSelect.options[i] = null;
           }
     }
}
```

The button call looks like:

```
onclick="DeleteOption(this.form, this.form.elements['bookmarkList']);"
```

Selecting All Options

Now if the user has a bunch of options, it could get very tedious to click through, say 20 bookmarks to delete them all. There is also one other very good reason to be able to select all the options in a SELECT object. In order to save the whole bookmark list the user has just changed around, **every option must be selected when the form is submitted**. ONLY the options that are *selected* will get passed along with the form object. If we fail to do this, only one, or even none of the user's bookmarks would be saved when the form was submitted. This would be a Very Bad Thing. Note this is only necessary when using a MULTIPLE select, which in this case we are.

So we'll want to use the function for two purposes, to help the user out, and to save the information correctly.

Again, we want to send the FORM and the SELECT object. And again we'll want to utilize the length of the options array and a FOR loop. This time we will do the traditional start at 0 and move up the ladder FOR loop. All we need to is set each option's selected property to true in the array, and every item will be selected:

```
function SelectAllOptions(oForm, oSelect)
{
     optLength = oSelect.length;
     for(i=0; i<optLength; i++)
     {
           oSelect.options[i].selected = true;
     }
}
```

The call from the button looks like:

```
onclick="SelectAllOptions(this.form, this.form.elements['bookmarkList']);"
```

If we wanted to then make sure all our options are selected when the form submits, we can call this same function using the **onsubmit** form handler. This event handler fires when a form is submitted:

```
<form name="bookmarks" action="someFormAction.php" onsubmit="
SelectAllOptions(this, this.form.elements['bookmarkList']);">
```

Notice the shorter **this** reference to the form – we don't need to say **this.form** since we are ON the form object itself. In which case, **this** means the actual form.

Final Code

The final HTML looks like:

```
<html>
<body>

<form name="bookmarks">
<h3>Irwin Fletcher</h3>
<h5>Add a Bookmark:</h5>
<label for="newURL">URL:</label> <input type="text" name="newURL" id="newURL"
/> <label for="newDescription">Description:</label> <input type="text"
name="newDescription" /> <input type="button" name="add" value="Add Bookmark"
class="btn" onclick="AddBookmark(this.form.elements['newURL'].value,
this.form.elements['newDescription'].value,
this.form.elements['bookmarkList']);" />

<h5>Bookmarks:</h5>
<strong>View: </strong><input type="radio" name="view" value="viewDescription"
id="viewDesc" checked="checked" /> <label for="viewDesc">Description</label>
<input type="radio" name="view" value="viewURLs" id="viewURL" /> <label
for="viewURL">URLs</label>

<div style="margin-top: 6px;">
<select name="bookmarkList" multiple="multiple" size="10">
     <option value="www.yahoo.com">Yahoo!</option>
     <option value="www.dmxzone.com">DMXZone</option>
     <option value="www.amazon.com">My Favorite Online Bookstore</option>
</select>

<input type="button" name="moveUp" value="Move Up"
onclick="MoveOption(document.forms['bookmarks'], 'up',
document.forms['bookmarks'].elements['bookmarkList']);" /><br />
```

```
<input type="button" name="moveDown" value="Move Down"
onclick="MoveOption(document.forms['bookmarks'], 'down',
document.forms['bookmarks'].elements['bookmarkList']);" /><br />

<input type="button" name="selectAll" value="Select All" class="btn"
onclick="SelectAllOptions(this.form, this.form.elements['bookmarkList'])" /><br
/>

<input type="button" name="edit" value="Edit Selected" /><br />
<input type="button" name="delete" value="Delete Selected"
onclick="DeleteOption(this.form, this.form.elements['bookmarkList']);" />
</div>

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

And the final JavaScript:

```
function AddBookmark(url, description, oSelect)
{
     var optLength = oSelect.options.length;

     if(!url)
     {
           alert("Please enter the URL for your bookmark.")
           return;
     }
     if(!description)
     {
           alert("Please enter the description for your bookmark.")
           return;
     }

     // check for duplicate values
     for(i=0; i<optLength; i++)
     {
           optionText = oSelect.options[i].text;
           valueText = oSelect.options[i].value;
           if(optionText.toLowerCase() == description.toLowerCase())
           {
                 alert("That bookmark already exists.")
                 return;
           }
           if(valueText.toLowerCase() == url.toLowerCase())
           {
                 alert("That bookmark already exists.")
                 return;
           }
     }
     oSelect.options[optLength] = new Option(description, url);
}

function MoveOption(oForm, upOrDown, oSelect)
{
     // The user's current selection
     var choice = oSelect.selectedIndex;
     // Where the user wants the option to go
     var newPosition;
     // Save the user's VALUE and TEXT values so they
     // can be copied to the new location.
     var oldVal = oSelect[choice].value;
     var oldText = oSelect[choice].text;

     // If the user has not made a choice, exit the function
     if (choice == -1)
     {
           alert("You must first select the item to reorder.");
           return false;
     }

     // Find out which way the user wants the option to go
     // Then change the the "choice" variable (the current selection)
     // accordingly.
     if(upOrDown == "up") newPosition = choice - 1;
     if(upOrDown == "down") newPosition = choice + 1;

     // Check to make sure there is an option to copy.
     // If at top, wrap to the bottom and vice versa.

     // make it last if its already at the top
     if(newPosition < 0) newPosition = oSelect.length - 1;

     // make it first if its already at the end
     if(newPosition >= oSelect.length) newPosition = 0;

     // Copy the new VALUE and TEXT values (above or below) to the
     // user's current location
     oSelect[choice].value = oSelect[newPosition].value;
     oSelect[choice].text = oSelect[newPosition].text;

     // Copy the user's VALUE and TEXT values to the
     // new location, finalizing the swap of values
     oSelect[newPosition].value = oldVal;
     oSelect[newPosition].text = oldText;

     // Move the selectedIndex to where the user
     // "moved" the OPTION
     oSelect.selectedIndex = newPosition;
}

function DeleteOption(oForm, oSelect)
```

```
{
     optLength = oSelect.length;
     for(i=optLength-1; i>=0; i--)
     {
           if(oSelect.options[i].selected == true)
           {
                 oSelect.options[i] = null;
           }
     }
}

function SelectAllOptions(oForm, oSelect)
{
     optLength = oSelect.length;
     for(i=0; i<optLength; i++)
     {
           oSelect.options[i].selected = true;
     }
}
```

Conclusion

We've only really scratched the surface of what can be done with a SELECT object. We could have gone further and given the user the ability to edit each individual bookmark while on the same page. And we also didn't get in to discussing pages with more than one select box and interaction between the two. Hopefully we'll cover those topics down the road.

Always remember to select all your options in a multiple select BEFORE you send your form information to the server, or bad data will rear its ugly head. If this happens, in the best case your back end will have minimal or no data. In the worst case, such incorrect data would cause errors and failure of the back end. Not a good thing.

The Date Object

Sooner or later you're going to find yourself in need of some kind of date in your scripts. In the "old days" using dates in JavaScript was hairy. However since then JavaScript's date object has come along quite nicely. When you add the fact that today our computers handle the time very efficiently with control panels (whether Mac or PC) by setting time zones and daylight savings, you have things well prepared for using dates in scripts.

Having said that things are better, using the date object IS tricky. And we have to remember that it is an OBJECT and treat it as such. In addition, you have to be somewhat aware of how time works around the globe. Particularly if the application you are writing is going to span time zones, because when a script runs on your users system, the date handles the LOCAL time.

Now that we have your attention, let's dive in!

Time Zones and the mysterious GMT

Most people realize that the time is not the same all over the world. The earth rotates and we all see the noon sun at a different time. Noon in Australia is not noon in the United States.

So there had to be some way that we could talk about time and understand that noon at one place is three o'clock somewhere else. This is where GMT or Greenwich Mean Time comes in. The standard reference point for time is the celestial observatory in Greenwich, England. If you take a slice of the world straight down through that place, you end up in the middle of the Pacific Ocean, and that "line" gives you the international date line.

So with GMT as the base, zones were created. I live in the Central Standard time zone, which is -6 from GMT. So I'm six hours behind GMT. You can find out how far away any time zone is from GMT by checking out the world clock at http://www.timeanddate.com/worldclock/. Simply find the closest city to you.

Now you might already be wondering, why worry about time – aren't we talking about dates? While we are primarily talking about dates, time is part of any date. In fact when you create a date in JavaScript the time is included unless you specify otherwise.

It's simply good to be aware of the zone differences and the fact that late or early in the day in one place might be the next (or previous) day in another place. For example, if you were building a complex date and time picker for an airline web site you would need to make sure you handled time zones correctly.

In fact, what we are going to end up building is a date picker (without the time element) that you can use for any application, and extend it as you see fit.

The Date Object is an... object

Before we get into specifics we should talk about the fact that the date object is an object. Yes, it sounds obvious and redundant. But in JavaScript we are used to working in an object-*based* manner but not a real object-*oriented* manner. Since date is a true object, we have to deal with it in that way.

The reality is that this provides some real nice benefits. Since date is a built-in JavaScript object, it comes along with some real nice properties and methods of which we can make use. This means we don't have to write them ourselves! That's good because dealing with the date can be tricky.

Creating a Date object

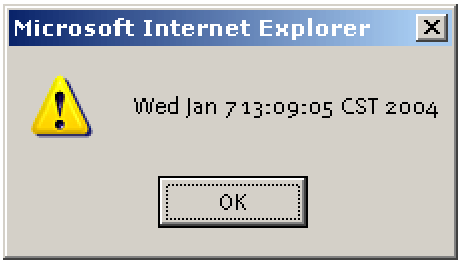
To create a date object we use the **new** keyword and the **Date** constructor. A constructor "constructs" the object with its default parameters. Those parameters are either built in by JavaScript (as is with the date object) or specified by you if you made your own custom object.
In the most basic terms we can make a date simply by doing the following:

```
var myDate = new Date();
```

Note that "**new**" is never capitalized, but the keyword "**Date**" *is* capitalized. The variable myDate now holds a reference to the date object you just created. If we alert that variable to see what it is, we get something like this (depending on what day you do it and what time zone you are in!)

```
<body>
<script type="text/javascript">
var myDate = new Date();
alert(myDate);
</script>
</body>
```

result:



```
Wed Jan 7 13:09:05 CST 2004
```

This alert came on Wednesday, 07 January 2004 at 1:09 in the afternoon, Central Standard Time. All that information was stored in the date object you created, and you have access to all of it via properties and methods. Now that's sweet.

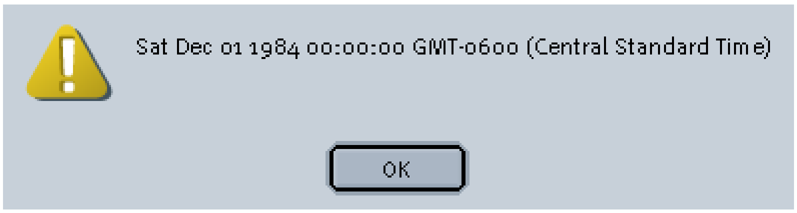
You could get more specific and decide exactly what your date object should hold. Take these three formats for example:

```
new Date(“Month”, “dd”, “yyyy”);
new Date(“yy”, “mm”, “dd”, “hh”, “mm”, “ss”);
new Date(“yy”, “mm”, “dd”);
```

If you were to set your parameters in any of these three ways, your date object would now only hold that information. For example, let’s do the third format, for Christmas day of the year 1984:

```
<body>
<script type=”text/javascript”>
var myDate = new Date(“84”, “11”, “01”);
alert(myDate);
</script>
</body>
```

This gives us:

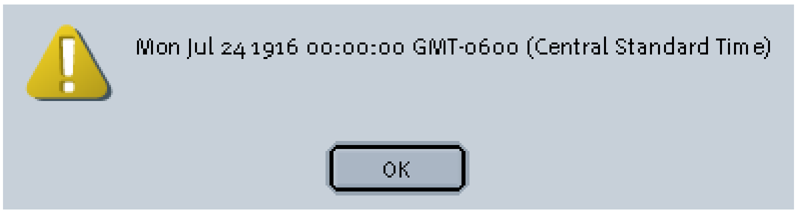


Sat Dec 01 1984 00:00:00 GMT-0600 (Central Standard Time)

But wait, you say – you put “11” for December! Yes, I did. That’s the old “arrays start with zero” JavaScript effect again. At any rate, this seems all fine and dandy, until we try to do the same date in the year 2001 in the first format, using the full 2001 numerals.

```
<body>
<script type=”text/javascript”>
var myDate = new Date("11", "01", "2001");
alert(myDate);
</script>
</body>
```

This results in:



Mon Jul 24 1916 00:00:00 GMT-0600 (Central Standard Time)

Whoops! 1916 it’s not! This is definitely a problem. Why doesn’t this work? Who really cares? There’s a much better way to deal with this. What we need is “setters and getters”!

Setters and Getters

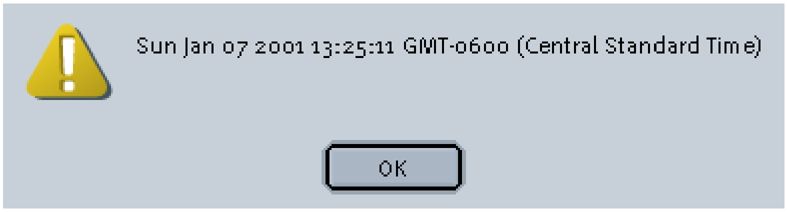
Setters and getters is the “in” term for functions or methods that set and get values. (What a creative term, eh?) As stated earlier, JavaScript has methods already in place for us to use for the date object. One of those methods allows us to, you guessed it, set the year. Examine the following:

```
<script type="text/javascript">
var myDate = new Date();
myDate.setYear("2001")
alert(myDate);
</script>
```

Notice the format of how method like this is used. First comes the object reference, and then comes the method name, followed by any supplied parameters. Thus:

```
objectName.methodName(parameters);
```

Is the correct format for using such methods. Now that we have that straight, let’s look at our result:



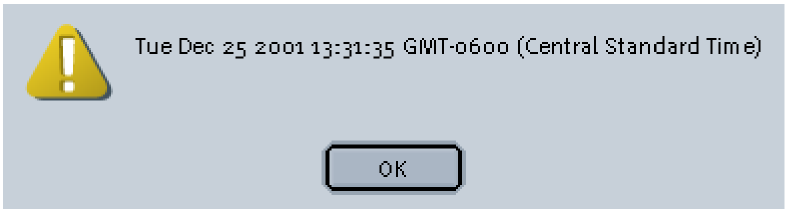
Sun Jan 07 2001 13:25:11 GMT-0600 (Central Standard Time)

Ah! 2001 just like we wanted! Honestly, we don’t really care why this way works and the other way doesn’t. Call it the Y2K bug if it makes you happy. Frankly, using the setters and getters is a better way of dealing with an object anyway.

Now, if we wanted to set the full Christmas date for that year, we can set the day and month as well using similar methods:

```
<body>
<script type="text/javascript">
var myDate = new Date();
myDate.setYear("2001")
myDate.setMonth("11");
myDate.setDate("25");
alert(myDate);
</script>
</body>
```

Which results in:



```
Tue Dec 25 2001 13:31:35 GMT-0600 (Central Standard Time)
```

Viola! The whole date as we intended. Notice that it also supplied the time as well. Now, I'll point out a couple gotchas to notice. First, as we just stated above note the "11" for December, not "12." The month array goes from 0 – 11. Secondly, yes the day retrieval method is indeed set**Date**. Why?

Surprisingly there's actually a fairly good reason. There is in fact two methods called **setDay** and **getDay**. However, these methods give you the day of the WEEK in which the DATE falls on. If you think about it, when you ask someone the date, they tell you "it's the 25th" or something like that, they don't tell you "it's Wednesday."

Now if you think about dates in calendar terms, the day a date falls on is vital. January 1st, 2004 falls on a Thursday. If we were to start our calendar on Wednesday instead, it would literally throw the whole year into a jumble.

The beautiful thing is, we won't have to overly concern ourselves with what day a date starts on for a given month because JavaScript knows that too! We'll just have to be careful that we wait until that day before we start writing out dates. That may sound a bit confusing, but we'll explain it better in the next tutorial.

Kick things around

Now what if we want to navigate to a previous or next month? Again we can use the setter methods to do this. Let's take our Christmas 2001 date and say we want to see the next month, which would be January 2002. Examine the following:

```
<body>
<script type="text/javascript">
var myDate = new Date();
myDate.setYear("2001")
myDate.setMonth("11");
myDate.setDate("25");

var nextMonth = new Date(myDate);
nextMonth.setMonth(myDate.getMonth()+1);

alert(nextMonth);
</script>
</body>
```

Now things can get confusing here so we'll take it step by step. First, note that nothing was changed in our previous code that gave us Christmas 2001. Secondly, notice we created a new date object (tied to the variable nextMonth) for our next month – but we sent the current Christmas date as the parameter.

The reason we're doing this is that we want to calculate one month previous of our Christmas 2001 date. So we need to base our calculation on that date, so we supply it. The new Date() constructor then can take any valid date reference as a parameter. It then just makes a duplicate of that date.

Now all that is left is to get the previous date. This will involve using a setter and a getter in one statement! Look again at the following line:

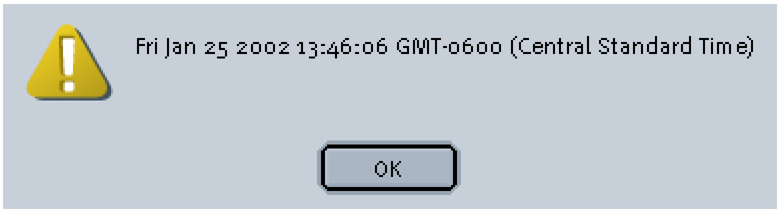
```
nextMonth.setMonth(myDate.getMonth()+1);
```

We're working on the nextMonth date, which right now is a copy of the Christmas 2001 date. We want to set that month to the previous one, so we'll use the setter. That is why we start with nextMonth.setMonth().

Then, inside the setter we're using the getMonth method. Remember the setMonth/getMonth parameters take a number from 0 – 11. So first we get the month, which ends up being 11 for December, because it's still our Christmas 2001 date (we haven't changed anything yet). Then we make our calculation – we add one to that month, thus the +1.

Now logically you might think that would give you 12, but in reality it gives you 0! That's because the month array is what you are dealing with here, and you just told the array to move to the next item in the array, which happens to be 0 for January.

So the result is:


Fri Jan 25 2002 13:46:06 GMT-0600 (Central Standard Time)

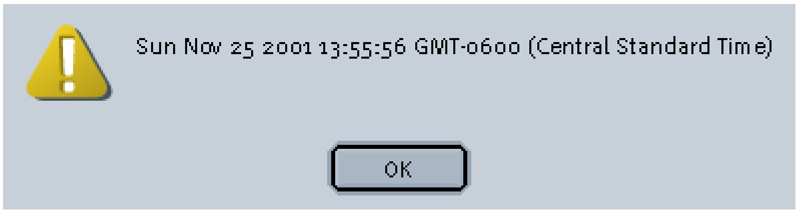
Perfect, Friday January 25th, 2002! One month later.

Now you can imagine we can get the previous month in the exact same manner, only subtracting one:

```
<body>
<script type="text/javascript">
var myDate = new Date();
myDate.setYear("2001")
myDate.setMonth("11");
myDate.setDate("25");

var prevMonth = new Date(myDate);
prevMonth.setMonth(myDate.getMonth()-1);
```

```
alert(prevMonth);
</script>
</body>
```

This would give us:



```
Sun Nov 25 2001 13:55:56 GMT-0600 (Central Standard Time)
```

One month previous.

Converting the date from the mm/dd/yyyy format

Now what if we wanted to convert that date from the common mm/dd/yyyy format into a date object? Well, you could use a lot of crazy regular expressions but we can actually avoid that. A simple function making use of the split method of an array and basic date formatting can do the trick.

```
function FormatDate(myDate)
{
     var dateValArray = dateValue.split("/");
     var currentDate = new Date(dateValArray[2],dateValArray[0]-
1,dateValArray[1]);
     return currentDate;
}
```

This little function takes a valid date reference as a parameter, such as our Christmas 2001 date. Now we're going to go slightly back on something said earlier, because I wanted you to be able to use the setters and getters. Passing the parameters to the date constructor as FULL year, month and then date seems to work fine, as you will see. We could also use the setters and getters for this two, but it's good to know a couple ways to do things.

The split method simply makes an array from a string, and it breaks up that string based on the character you supply. Since the slash is the divider in our date format, we supply the slash. That gives us the following array:

```
dateValArray[0] = “12”
dateValArray[1] = “25”
dateValArray[2] = “2001”
```

The nice thing is we didn't have to specify that because the split method did it for us! So now we make a new Date constructor and using the correct places of the array, place the values in. Since the 2 position holds the full year we put that in first, then the month. Notice the -1! Remember our 0 –

11 array! And finally the 1 position for the date. We then return the currentDate variable and we have a valid date object again!

This will come in handy next time!

Conclusion

While the date object can be complex, it is very powerful. The basics are actually fairly simple to use, especially since JavaScript can do a lot of the dirty work for you.

Next column will build on the concepts above and make a date picker that you can use for any application. You'll be able to browse through the months, pick a date and have it displayed in a form control.

Forms

Scripting Forms

Interacting with forms has caused many a developer to tear out their precious hair in frustration. Luckily, it doesn't have to be that way. Many of the troubles that face the developer when scripting a form are syntactical. In addition, many people don't realize the power that JavaScript puts at their fingertips with tools like the elements[] array and the type properties. Correctly using these tools will make your life much easier.

Job One: Correctly Referencing Elements

One thing you will learn quite quickly in JavaScript is that you can't do anything if you aren't correctly referencing the object you wish to script. I dare not count the endless hours I spent as a beginner simply trying to figure out how to correctly "point" at something.

With forms and form elements, your best friend is the **DOM level 0**. But what is the DOM? There are lots of really geeky technical explanations on the web you can read for days explaining the intricacies of the DOM. Well, after you are done with that, we can simply explain it in one simple sentence. **The Document Object Model gives us access to every element in a document.** There, that was easy, wasn't it? Having access to every element makes life much simpler.

There are 4 levels of the DOM that exist, from DOM level 0 all the way up to the current level 3. Forget all that for right now. DOM level 0 has been in existence since Netscape 2 and Internet Explorer 3, and is supported by all modern browsers. So browser compatibility with your scripts has become a non-issue. Yes, that is a Good Thing(tm).

Getting the Form

The first thing you always need to do when scripting forms is **reference the form** in which your elements reside. If you have more than one form on a page, this becomes even more important.

The DOM contains an array of every form on the page. If you are not sure what an array is, it is quite simple. An array is used to store a series of related data items. You can place as many items in an array as you wish, and you can reference them by both the position in the array (give me the third item) or its name or "key" value (give me the item "address").

Array example

Think of an array as a simple html table. The table has a title, "robots." The table has two columns and three rows. The first column is the "key" values and the second column is the "value" values. Let's take a look:

```
robots

+--------+---------+
| red    | blappo  |  [0]
+--------+---------+
| green  | zappo   |  [1]
+--------+---------+
| blue   | zippi   |  [2]
+--------+---------+
```

Each key value in the left column is associated with the value in the corresponding right column. So if I wanted the green robot's name, I would get "zappo" as my result. I can also reference the right column values by the position. Remember JavaScript arrays start with 0, not 1 (note the bracketed values to the right of the table). So I could ask for the name of the robot in position 2, and I would get "zippi."

Translating this to actual code is now simple. We have an array called robots:

robots[];

If I wanted to place "zappo" into a variable from my robots array, I would write the following statement:

```
myRobotName = robots['green']; or
myRobotName = robots[1];
```

Both would return "zappo" as our name. You can learn more about how to create and manipulate arrays from your favorite resource, but this is enough to move on with.

Getting the Form

So remember, we have an array of every form on the page, called forms[]. If you are a good webmonkey, you have named your form. Let's use the following HTML snippet as our code for the rest of this paragraph:

```
<form name="newRobots">
New Robot Name: <input type="text" name="newRobot" />
Robot Description: <textarea name="robotDescription"></textarea>
Robot Category:
<select name="robotCat">
     <option default="default" value="none">Choose a category:</option>
     <option value="scary">Scary</option>
     <option value="friendly">Friendly</option>
</select>
Will this robot rust?
<input type="radio" name="robotRust" value="yes" checked="checked" /> Yes
<input type="radio" name="robotRust" value="no" /> No
```

```
Check the box if you would like this robot to attack your home:
<input type="Checkbox" name="robotAttack" value="attackMe" />
<input type="submit" name="submit" value="Add Robot" />
</form>
```

Now, getting our form is a simple matter of the following:

```
myForm = document.forms['newRobots'];
```

Easy, right? You could also do:

```
myForm = document.forms[0];
```

We talked about self-describing code. This doesn't tell you much about your code at all. In addition, should you ever add another form on your page that comes before the newRobots form, your code would break.

Now that we have our form referenced in the variable myForm, we can use that reference throughout any script without having to constantly write out "document.forms['newRobots']" over and over. This is a good practice to follow.

Getting to the Elements

Now let's assume for some reason we wanted to get the value of form element (possibly for validation purposes, or other reasons.) If we follow the order of the form, the new robot name is first.

Just like there is a forms[] array, there is also an elements[] array for each form on your page. This elements[] array contains every element inside the form you specify. You now attach the elements array to the forms array like so:

```
forms['newRobots'].elements[];
```

Of course, we can use our variable:

```
myForm.elements[];
```

Now to specifically get our text box:

```
newRobotName = myForm.elements['newRobot'];
```

Again, we use the name (literally, the name attribute of the form element) to access the item we want in the elements array. Now our variable newRobotName is a reference to our text box. However, we want the actual text value, not just the text box. The text box element has a "value" property, so we can access that to get our new robot name:

```
newRobotName = myForm.elements['newRobot'].value;
```

And voila! You have the text that the user input into that text field. Getting the value of the textarea is exactly the same, just reference the textarea instead of the text box:

```
newRobotDescription = myForm.elements['robotDescription'].value;
```

But what about the robot category? What if we were to do:

```
newRobotDescription = myForm.elements['robotCat'].value;
```

Believe it or not, that would indeed be correct. If you look at how a select box is written, you'll notice that each option element has a value. If you were diligent in setting a value for each option, the select box will gladly return the value of the selected item. Scripting the select box can get tricky when you attempt more complex tasks, but simply getting the selected value is as simple as the text box.

Now comes our first special case, radio buttons. If you are paying close attention you have just realized that both radio buttons have the same name. This is because radio buttons are set up as a group, and only one radio button can be selected at a time. Each radio button does indeed have its own unique value, but if we were to attempt the following:

```
willRobotRust = myForm.elements['robotRust'].value;
```

It would result in an 'undefined' error. So what gives? The fact is that robotRust is a group, not an individual item. We need to get one step deeper. Ironically, we are able to use yet another array to do that! A radio group has an array that contains each button. Strangely, this array does not have a name, and referencing it looks kind of strange. Remembering that arrays start with 0, if we wanted to get the value of the first button, we would do:

```
willRobotRust = myForm.elements['robotRust'][0].value;
```

Or to get the second button:

```
willRobotRust = myForm.elements['robotRust'][1].value;
```

But this isn't helping much, you want to get whichever value is selected so you know whether or not the robot will rust. With a case of only two radio buttons, a simple if then condition will help:

```
var willRobotRust;
var rustRadioButtons = myForm.elements['robotRust'];

if(rustRadioButtons[0].checked)
{
     willRobotRust = "yes";
}
else
{
     willRobotRust = "no";
}
```

With more than two or three buttons, an if else condition becomes a bit clumsy. Let's imagine we had a radio group with 10 buttons, and we wanted the value of the checked button. Any array has a property called **length**. This gives you the number of items in the array. The gotcha here is again to remember that JavaScript arrays start with 0, so in this test case our length is going to be nine, not ten! Then using a for loop, we can find out which button is checked. We'll loop through (you guessed it!) the unnamed array of buttons in our radio group and test each button in the group to see if it is checked. When we find the one that is, we can set our variable to its value.

```
var radioButtonValue;
var radioButtons = myForm.elements['theRadioGroup'];

for(i=0; i<radioButtons.length; i++)
{
     if(radioButtons[i].checked == true)
     {
           radioButtonsValue = radioButtons[i].value;
           return;
     }
}
```

If you aren't familiar with a for loop, it's not as difficult as it may look. A for loop takes three parameters separated by semicolons:

1. i=0; - i is simply a variable name you choose, and i happens to be the traditional choice. You can start with any value, but here we start with 0, since arrays start at 0 as well.

2. i<radioButtons.length; - this is the 'while' condition of the for loop. It tells the loop how long to continue. In our case, we are saying "as long as i is less than the total number of radio buttons, keep going." The length property attached to our radio button group (radioButtons) gives us our number, which is nine, as we discussed above.

3. i++; this is the increment value. In our case (and probably in most cases) you want to step through one by one. i++ is shorthand for i = i + 1.

Now, inside our for loop is an if condition. A property of a radio button is whether or not it is checked, and if you have coded your HTML correctly, only one radio button in a group can be checked. As you can see we are using the array that holds each radio button attached to our variable holding our radio group, radioButtons[i]. The variable i starts as 0, and continues all the way to 9 (if we were to go all the way through) in our case. So the first time it is radioButtons[0], the second radioButtons[1] and so on.

If that radio button happens to be checked, we jump inside the if clause and you can see that now we can actually get the checked radio button. Since the checked condition evaluated as true, we know that the radio button at radioButtons[i] is the one selected, so we grab its value.

Now don't get confused here, we are grabbing the value of a single button - not the radio group. If we were to write it out fully, it would be:

```
radioButtonsValue =
document.forms['someForm'].elements['theRadioGroup'][i].value;
```

Because we used shorthand as it were, myForm for our form reference and then radioButtons for our radio group, it made the reference shorter. This is nice, but I want to make sure you realize just where we got the value.

Checkboxes

Lastly, we have a checkbox. Again, you can be tripped up here if you don't be careful. If we were to set the following variable and test it:

attackedValue = myForm.elements['robotAttack'].value;

We would find that it will always return 'attackMe', regardless of whether or not it is checked. Why? Because, that happens to be the value of the checkbox. Whether it is checked or not doesn't change the value.

In this particular case, we are using the checkbox to determine a boolean (that is, a **true** or **false**) value. We want to know whether or not we should be attacked, based on whether or not the checkbox is checked. Once again, a simple if else condition and a check of the checked property will give us what we need:

```
var attackCheckbox = myForm.elements['robotAttack'];
var attack;

if(attackCheckbox.checked == true)
{
     attack = "yes";
}
else
{
     attack = "no";
}
```

So in this case, the value of our checkbox is effectively meaningless. But we can take action based on whether it is checked or not.

There are certainly many cases where you would use more than one checkbox if a user has the choice of making more than one choice. We'll leave more complex situations for another time.

Conclusion

As you can see, using the DOM level 0 makes it pretty simple to access the values in a form that you need. As a bonus, it's completely cross-browser compatible and simple to execute. You should now be able to attempt more complex scripts than you ever thought possible!

JavaScripting usable forms

So you're looking for that rare part to fix your widget-stick, since it's been broken for nearly a month, and you finally come across widgetstick.com. They happen to have just the part you need. Ecstatic, you add your part to their shopping cart and begin the checkout process.

As you are filling out your personal information, you notice the next section, billing information. You groan realizing that you now have to type all the information twice. In vain you look for some functionality on the page that will avoid this (and in fact, you attempt to submit the page anyway, but get an error) without any luck.

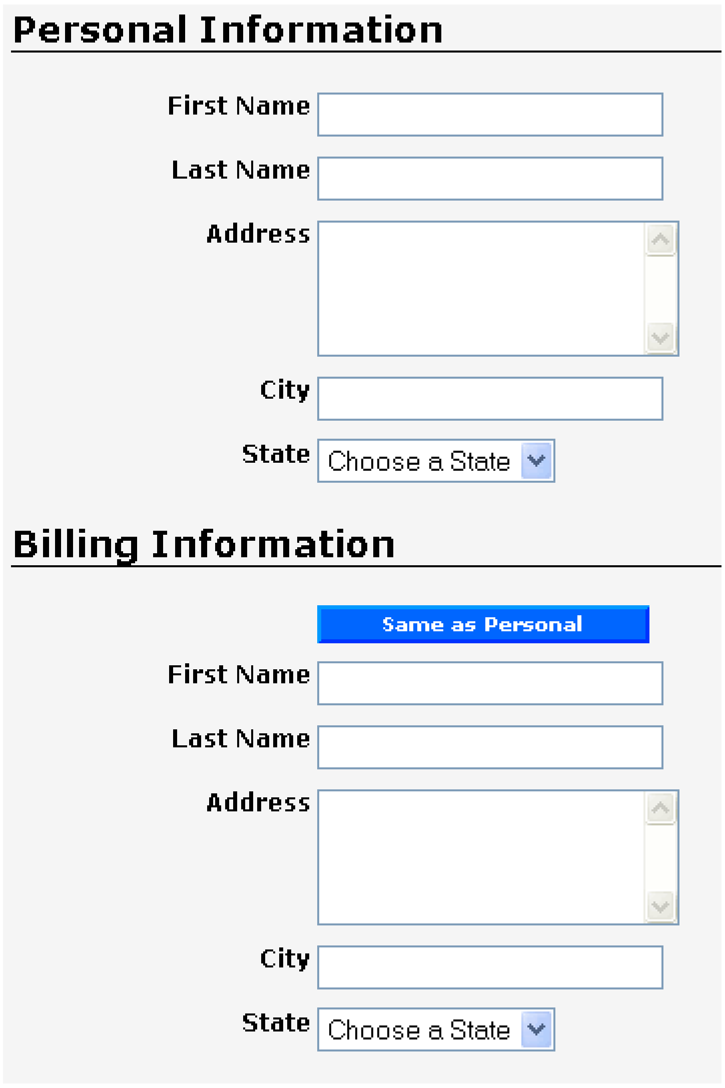
If only they had one of those "copy personal information to billing" buttons you think, like other well built sites…

What a great idea! Let's build one.

The Components

This type of functionality is something that definitely makes life easier for your users/customers. This is, of course a Good Thing™.

The first thing we need is an example form. The screen shot below and the accompanying HTML will be the base with which we work. Note there is no submit button, we'll just be working with the "Personal Information" and "Billing Information" sections of the form.


Personal Information

First Name

Last Name

Address

City

State Choose a State

Billing Information

Same as Personal

First Name

Last Name

Address

City

State Choose a State

Notice that we have three types of form elements that will need to be dealt with when copying the information: text field, textarea (for the address) and a select box. This is a pretty basic form, and for clarity and simplicity I've left off the typical zip code, phone number and email address fields (in addition to any others that might normally appear) as well as only listed three states in our select box.

The button marked "Same as Personal" will be the trigger for the user to copy all the information they just entered from the top section to the bottom section, thus saving them keystrokes which would have undoubtedly led to groans and mutterings. We cannot have our customers groaning and muttering!

Below is the HTML for the form:

```
<form name="billing">
<h3>Personal Information</h3>
<label for="fname">First Name</label> <input type="text" name="fname"
id="fname" size="25" />
<label for="lname">Last Name</label> <input type="text" name="lname" id="lname"
size="25" />
<label for="address">Address</label> <textarea name="address" id="address"
cols="20" rows="4"></textarea>
<label for="city">City</label> <input type="text" name="city" id="city"
size="25" />
<label for="state">State</label>
<select name="state" id="state">
     <option value="0">Choose a State</option>
     <option value="ID">Idaho</option>
     <option value="IL">Illinois</option>
     <option value="IN">Indiana</option>
</select>

<h3>Billing Information</h3>
<input type="button" name="copy" value="Same as Personal"
onclick="CopyToBilling(this.form);"  /> <label for="fnameB">First Name</label>
<input type="text" name="fnameB" id="fnameB" size="25" />
<label for="lnameB">Last Name</label> <input type="text" name="lnameB"
id="lnameB" size="25" />
<label for="addressB">Address</label> <textarea name="addressB" id="addressB"
cols="20" rows="4"></textarea>
<label for="cityB">City</label> <input type="text" name="cityB" id="cityB"
size="25" />
<label for="stateB">State</label>
<select name="stateB" id="stateB">
     <option value="0">Choose a State</option>
     <option value="ID">Idaho</option>
     <option value="IL">Illinois</option>
     <option value="IN">Indiana</option>
</select>
</form>
```

As usual we are using the LABEL tag for accessibility and usability. We've not used the TABINDEX here since tabbing through this form is straightforward, in the order it is built, and that works just fine.

TIP: You may notice I have not put my forms in my examples into tables for layout. I am also using things like underlines below my headers and some color and font styling. This is all done with CSS. That is why I don't need to use the <br> tag or tables. Certainly, tables can be and are quite often a logical and acceptable way to lay out forms. My main goal is to keep the HTML in the examples clean so you can understand what is going on easily. If you are interested in these techniques one good paragraph is Practical CSS Layout by Mark Newhouse at http://www.alistapart.com/stories/practicalcss/.

Building the Script

Parameters

As usual when dealing with a script that manipulates a form, we like to send the form as a parameter to make things easier. We've talked about abstracting functions recently, but in this case the information is so specific to this page that that isn't going to be possible. In any event, the form object sent as a parameter will make things much easier.

```
function CopyToBilling(oForm)
{

}
```

And as mentioned, we'll call this function form the button at the top of the billing information:

```
<input type="button" name="copy" value="Same as Personal"
onclick="CopyToBilling(this.form);"  />
```

Local Variables

We've got eleven objects that we need to reference, five fields in each section of our form and the button we're using to trigger the function. It might seem like a lot to lay out eleven variables at the top of our script. However, sometimes with a very specific script like this one, there isn't a good way around it. You could try to get fancy and possibly set up a couple of arrays, but in the end you'd be writing too much code. It's easier to simply declare your objects as variables and be done with it.

Since we already have our form reference as a parameter, referencing each field is simple. Take the personal first name as an example. We'll preference each personal information field with a "p", use the form object and the elements array and build each variable:

```
var pFirstName = oForm.elements['fname'];
```

It's quite simple. We'll need 10 more of these, but we should at least use some white space to give them some separation for readability. We'll also use "b" as the prefix for the billing fields. When we are done the variable list (with a couple well placed comments) should look like:

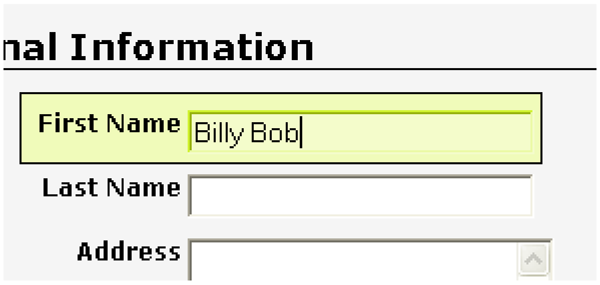
```
// Personal Info Fields
var pFirstName = oForm.elements['fname'];
var pLastName = oForm.elements['lname'];
var pAddress = oForm.elements['address'];
var pCity = oForm.elements['city'];
var pState = oForm.elements['state'];

// Billing Info Fields
var bFirstName = oForm.elements['fnameB'];
var bLastName = oForm.elements['lnameB'];
var bAddress = oForm.elements['addressB'];
var bCity = oForm.elements['cityB'];
var bState = oForm.elements['stateB'];

// Button to toggle script
var toggle = oForm.elements['copy'];
```

Remember that using the VAR keyword **inside** a script confines those variables have meaning **only within the script in which they reside**. They would have no meaning outside this script. They would be considered "out of scope."

Now we have everything we need to make the copy. We've got three different types of form elements to deal with: the text field, the textarea and the select box.

Copying the Text Fields

The text fields are going to be quite simple. Remember back when we looked at the form elements in the "Scripting Forms" paragraph we talked about each objects properties and methods. The property we are after for the text field in this case is **value**. The value property refers to the VALUE attribute of the element. So if we saw this on the screen:



The value property for the "First Name" field is equal to "Billy Bob." We could change that on the fly with a script if we wanted. Say we wanted the field to say "Sally Jo" instead. We could simply do:

```
var myTextField = oForm.elements['fname'];
myTextField.value = "Sally Jo";
```

And the field would take on the new value. This is all we have to do to copy our text fields. To copy the personal first name to the billing first name, all we need to do is:

```
bFirstName.value = pFirstName.value;
```

Simple! Now this might seem like an awfully short line to accomplish copying one form element's information to another. But remember we did all the hard work already when we specified all the form elements as variables. From here on out, simply using the variable name and its' attached properties is all we need to manipulate the fields.

So in this case, we said take the billing first name field's **value** (`bFirstName.value`) and make it equal to the personal first name field's **value** (`pFirstName.value`). There you have it, value copied. So for the other text fields (last name and city), we do the same thing:

```
bLastName.value = pLastName.value;
bCity.value = pCity.value;
```

Copying the Textarea

Next comes the textarea. There must be some special operation we need to do here, since a textarea is different from the text field, right? Wrong! It is the same property, **value**, that is used. So we have the same type of line:

```
bAddress.value = pAddress.value;
```

Copying the State

Ahhh, but copying the state – THAT will be difficult you say! Only slightly more so. Remember our good friend the **selectedIndex** property. He tells us which option in a single select box is currently selected by the user (or the first selection in a multiple select). The first thing we need to do is get that value. Once we have that value, we can tell the billing select box to have the same **selectedIndex**, and they will match.

If you remember back to the past few paragraphs on the select object, **selectedIndex** will give us the array position the current selection resides in the OPTIONS array. Yes.. you knew it was coming… the first option is indeed "0."

We actually need one more local variable, which you may have noticed we left out in the beginning. We need the current selection of the state in the personal information section. We get this like so:

```
var selectedState = pState.selectedIndex;
```

This gives us that array number which we can use to set the billing state select box. But wait – we should make sure the user has selected a state first. If they haven't then all we would copy over simply the "Choose a State" selection. Then when the user does submit the form, they will get an error. That's simply bad programming.

In the case of this single select box, if the user has not chosen a state, the value of selectedState will be – you guessed it – 0. We can use a simple IF control structure to check for that value, and if it is true we can alert the user and tell them to choose a state. We'll use the nifty return statement to exit out of the function before making the copy. Then they can fix the problem and hit the button once again.

Of course, we need to do this **before** copying any fields.

```
if(selectedState == 0)
{
     alert("Please choose a state.");
     return;
}
```

Remember that the double equals operator checks for equality, while the single equals assigns a value. So we use the double in this case. If it is in fact 0, the user is told to select a state and we return out of the function without copying anything.

If that is all well and good, the only thing left to do is make the billing select box reflect the same choice (same **selectedIndex**) as the personal. Again, we've set up all the variables we need to do that, so it's pretty simple to do:

```
bState.selectedIndex = selectedState;
```

We've merely told the billing select box to have the same **selectedIndex** as the personal select box (which is stored in the selectedState variable.)

Our whole function looks like:

```
function CopyToBilling(oForm)
{
     // Personal Info Fields
     var pFirstName = oForm.elements['fname'];
     var pLastName = oForm.elements['lname'];
     var pAddress = oForm.elements['address'];
     var pCity = oForm.elements['city'];
     var pState = oForm.elements['state'];

     // Billing Info Fields
     var bFirstName = oForm.elements['fnameB'];
     var bLastName = oForm.elements['lnameB'];
     var bAddress = oForm.elements['addressB'];
     var bCity = oForm.elements['cityB'];
     var bState = oForm.elements['stateB'];

     // Selected state
     var selectedState = pState.selectedIndex;

     // button to toggle script
     var toggle = oForm.elements['copy'];

     // check for valid state selection
     if(selectedState == 0)
     {
           alert("Please choose a state.");
           return;
     }

     // copy fields
     bFirstName.value = pFirstName.value;
     bLastName.value = pLastName.value;
     bCity.value = pCity.value;
     bAddress.value = pAddress.value;
```

```
      bState.selectedIndex = selectedState;
}
```

Conclusion

As you can see simply copying values from one element to another is relatively straightforward. As usual, passing the form object as a parameter to the form and setting up our objects as local variables makes things go quite smoothly.

We couldn't really abstract this function, but that's okay. As I said before, not every function can be abstracted, especially when they are as specific as this one. There might be other ways to work this function as well – for example you could send along the selected index of the personal state selection along at the function call as a parameter. Sometimes its just a matter of preference. Play with the script on your own to see what variations you can come up with, maybe even adding a few new fields to make things more interesting.

JavaScript: Disabling and Enabling Form Fields Dynamically

Sometimes you have a form that has dependencies. By this I mean that one (or more) field's values depend on another field's values. For example, you may not want the user to fill in field B until the user has filled in field A.

One way to solve this is to dynamically hide and show form fields, which we have talked about in the past. Some people don't like parts of their form disappearing and reappearing on them, however. If that is the case what you can do is disable and enable them instead.

Disabled vs. ReadOnly

There are two similar attributes of a form input that can address this issue. The first one is the attribute readonly. Readonly allows you to display a value without allowing the user to modify it. In many cases this would suit your purposes well, but we are considering a new form, not displaying results from an existing form.

The disabled property gives us something more, in that it cannot receive any events, or focus in the browser (by user or by script - however any text in a disabled element can be copied and pasted, like in a readonly element.) One important thing to note as well is that a disabled form field will NOT send its data on form submission. This might be another helpful reason to use the attribute should you not want to send data from a particular field for some reason.

Disabled also sometimes gives an element a "dimmed" look. While this is a nice feature, it varies on how it looks from browser to browser, so we'll use our own styles to get the look we want.

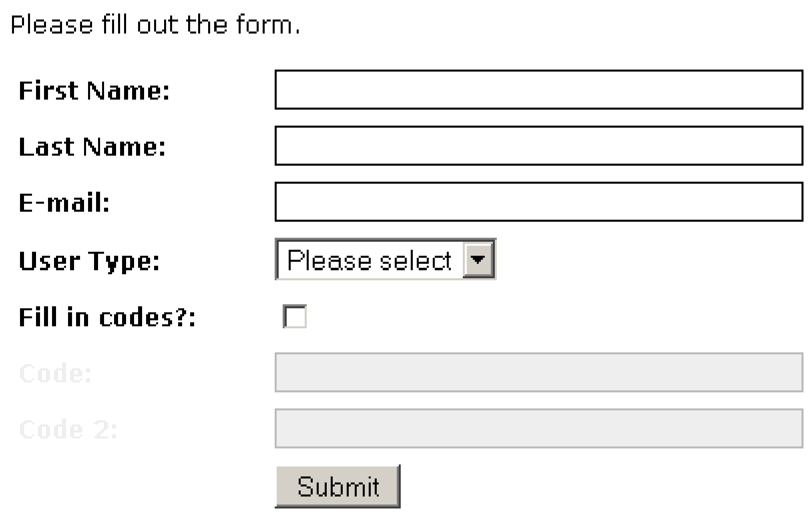
The HTML Form

Let's begin with a simple HTML form, for which the code is below:

```
<body>

<p>Please fill out the form.</p>

<form name="myform">

<table border="0" cellspacing="0" cellpadding="4">
<tr>
     <td width="120" class="inputLabel"><label for="name_l"
class="labelEnabled"> First Name:</label></td>
     <td><input type="text" name="name" id="name_l" size="40" class="textBox"
/></td>
</tr>
<tr>
     <td class="inputLabel"><label for="last_l" class="labelEnabled">Last
Name:</label></td>
     <td><input type="text" name="last" id="last_l" size="40" class="textBox"
/></td>
```

```
</tr>
<tr>
     <td class="inputLabel"><label for="email_l" class="labelEnabled"> E-
mail:</label></td>
     <td><input type="text" name="email" id="email_l" size="40" class="textBox"
/></td>
</tr>
<tr>
     <td class="inputLabel"><label for="type_l" class="labelEnabled">User
Type:</label></td>
     <td>
           <select name="type" id="type_l" onchange="toggleFields(this.form,
'code', 'code2'); ">
                 <option value="">Please select</option>
                 <option value="MC">Management</option>
                 <option value="Candidate">Candidate</option>
                 <option value="Coach">Coach</option>
           </select>
     </td>
</tr>
<tr>
     <td class="inputLabel"><label for="fill_l" class="labelEnabled"> Fill in
codes?:</label></td>
     <td><input type="checkbox" name="fill" id="fill_l"
onclick="toggleFields(this.form, 'code', 'code2');" /></td>
</tr>
<tr>
     <td class="inputLabel"><label for="code_l" id="code_l"
class="labelDisabled">Code:</label></td>
     <td><input type="text" name="code" id="code_l" size="40"
disabled="disabled" class="TextBoxDisabled" /></td>
</tr>
<tr>
     <td class="inputLabel"><label for="code2_l" id="code2_l"
class="labelDisabled">Code 2:</label></td>
     <td><input type="text" name="code2" id="code2_l" size="40"
disabled="disabled" class="TextBoxDisabled" /></td>
</tr>
<tr>
     <td>&nbsp;</td>
     <td><input type="submit" value="Submit" /></td>
</tr>
</table>

<input type="hidden" name="hidSub" value="0"/>
</form>


</body>
```

Nothing too complicated, the form displays like the screen shot below:


Please fill out the form.

First Name:	
Last Name:	
E-mail:	
User Type:	Please select
Fill in codes?:	☐
Code:	
Code 2:	
	Submit

Notice that the last two text fields are grayed out, along with their associated labels. The user cannot type any text into the fields or give them focus.

The CSS associated with the file is below:

```
<style type="text/css">
body {
     font-size: 12px;
     font-family: verdana, sans-serif; }

.labelEnabled {
     text-align: right;
     font-weight: bold;
     color: #000; }

table {
     font-size: 12px; }

.TextBoxDisabled {
     background: #eee;
     border: 1px solid #bbb; }

.textBox { background: #fff; border: 1px solid black; }

.labelDisabled {
     text-align: right;
     font-weight: bold;
     color: #eee; }
</style>
```

The Script

What we want is some kind of function that will toggle the fields when our "trigger" field is filled in or manipulated. Then when the user makes a change, we enable the fields. The event we can use to trigger this is the **onchange** event.

Onchange is cool because it looks for a different state than what was there when the page loaded. So if you change the drop down menu – the event handler will fire. If you change it back, it will kind of "undo" itself, as you will see shortly.

First let's get a look at the script:

```
function toggleFields(oForm)
{
     for(i = 1; i < arguments.length; i++)
     {
           var field = oForm.elements[arguments[i]];
           if(field)
           {
                 var oLabel = field.name + "_l";
                 if(field.disabled == true)
                 {
                       field.disabled = false;
                       if(field.type == "text") field.className = "textBox";
                       document.getElementById(oLabel).className =
"labelEnabled";
                 }
                 else
                 {
                       field.disabled = true;
                       if(field.type == "text")
                             field.className = "TextBoxDisabled";
                       document.getElementById(oLabel).className =
"labelDisabled";
                 }
           }
     }
}
```

Notice that the only parameter we send is the form that contains the fields that we want to manipulate. That may seem odd (how do you know what fields to manipulate?), but I will explain in a moment. Notice also that we are changing class names as we go through, so that when the fields are enabled they look editable, and vice versa. We do the same for the actual form field labels, too, which is a nice added bonus.

Calling the function

Let's first look at an example of the function call in action:

```
<select name="type" id="type_l" onchange="toggleFields(this.form, 'code',
'code2');">
```

On our SELECT menu we use the **onchange** event handler as discussed above. Because the SELECT resides in the form we are going to manipulate, we can use the **this** keyword to reference the form – **this.form** – in other words "the form which I (the SELECT) reside in."

And then you see we are sending two more parameters – but wait! I said there was only one parameter! Yes, and this is where we can get tricky. We never can know how many fields we want to enable, and we'd like to keep this function as abstract as possible. Therefore we can build it to allow toggling of one, or two or even 10 fields. So for each field you want to toggle, you send the NAME of that field as a parameter. We'll use the magic of the arguments array to pull them out.

The arguments array

Remember that parameters are also called arguments and vice versa – they are the same thing. JavaScript has a nice built in object called the arguments array that is available to any function you create. The arguments array is exactly what it says – it's an array of all the arguments sent to the function. We can see it in action in the following line – the first line of our function:

```
for(i = 1; i < arguments.length; i++)
```

We are going to loop through each parameter/argument sent to the function, and handle each one. Notice however that we start our FOR loop at ONE not ZERO! Normally we would start our loops with zero, but in this case the first parameter is known to us – it's going to be the form object (this.form). We know that is not a form *field* and we don't want to perform our statements on it, as it will generate errors.

In this case the arguments array is going to be three. So we are going to loop through 2 and 3, and those values are going to be 'code' and 'code2'.

Now that we have that set up we do the following:

```
var field = oForm.elements[arguments[i]];
     if(field)
{
```

We set a variable called field using the elements array just like we would normally get a form field. But of course, we want to use the field names supplied by the function call (code and code2), so we are going to put the current item of the arguments array in the spot where we would normally place the name. That would be:

```
arguments[i]
```

This will evaluate to "code" the first time through and "code2" the second time through. You can see how nice it is to be able to use this and never have to worry about how many parameters you need to deal with. The arguments array can be very powerful for many situations like this.

So doing the above is the same as doing:

```
var field = oForm.elements['code'];
```

And so forth. The variable field now holds the text field we want to now enable. Notice that before we step in and start manipulating the field, we check for its existence using **if(field)** to make sure the script sees the field. If it didn't see it due to some error (a mis-spelling in our function call for example) then at least the user won't get an error – the code inside the block will fail to execute. True, the user will not get his field enabled, but you avoid showing messy JavaScript errors to the user, which is even worse.

Now we want to set up a variable to handle the associated label of our form field. You may have noticed that we named our labels the exact same name as the text fields, except we added a "_l" to the end. Since we know that, we just build the name of the current label like so:

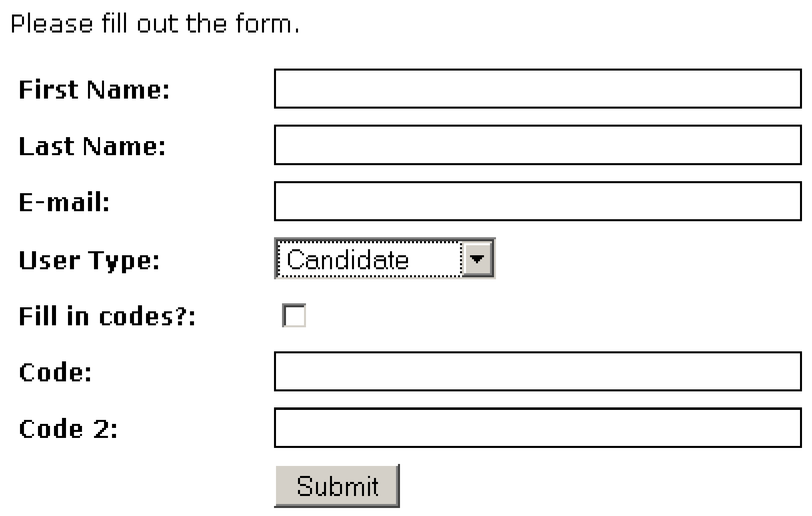
```
var oLabel = field.name + "_l";
```

Next we want to check and see if the field is disabled, which is a read/write property. This gives us the toggling functionality of the function. If it already is disabled, we'll step into the first IF condition:

```
if(field.disabled == true)
{
     field.disabled = false;
     if(field.type == "text") field.className = "textBox";
     document.getElementById(oLabel).className = "labelEnabled";
}
```

If the field is already disabled, we set disabled to false, thus enabling the field. Next we check for the field TYPE. Remember every form element has a type – checkbox, text, password, etc. If this field is a text box we set the class to a new class called **textbox** which shows the field as enabled (not grayed out anymore). You could expand this function to handle styles for more types of form elements if you wanted to.

Lastly, we grab the associated label and change its class to one that shows the label as enabled as well. So after changing the SELECT menu, the user would see the following:



Now the fields are enabled! Notice the drop down now says "candidate" instead of "select one." The fields **code** and **code 2** are now editable. If the user were to change the drop down back to the default position, the fields would then become disabled again. (Note: If you type some data into one of the enabled fields and then disable the fields by returning the drop down back to default, the text stays in the field – but it doesn't get sent by the form as described at the beginning of the paragraph.)

This toggling effect is handled in the ELSE clause:

```
else
{
     field.disabled = true;
if(field.type == "text") field.className = "TextBoxDisabled";
     document.getElementById(oLabel).className = "labelDisabled";
}
```

This is merely the opposite of what has happened in the IF clause. Disabled is now set to true, and we apply the disabled versions of the classes so the labels and fields appear disabled.

You probably noticed that the function call is also on the checkbox as well. Checking and unchecking the checkbox will give you the same result as using the drop down menu, so you can trigger this function from a checkbox as well.

Conclusion

Having a bit more control over your forms is always helpful and this can help you protect or cordon off part of your forms until certain tasks have been accomplished. Just remember to always validate your forms, as strange things are always just around the corner when it comes to HTML forms – good validation is a must.

You could easily modify this form to do many things. See if you can figure out how to disable/enable a whole form using this function as a base.

Advanced Forms

This chapter shows you how to extend and improve your usability and accessibility of your forms.

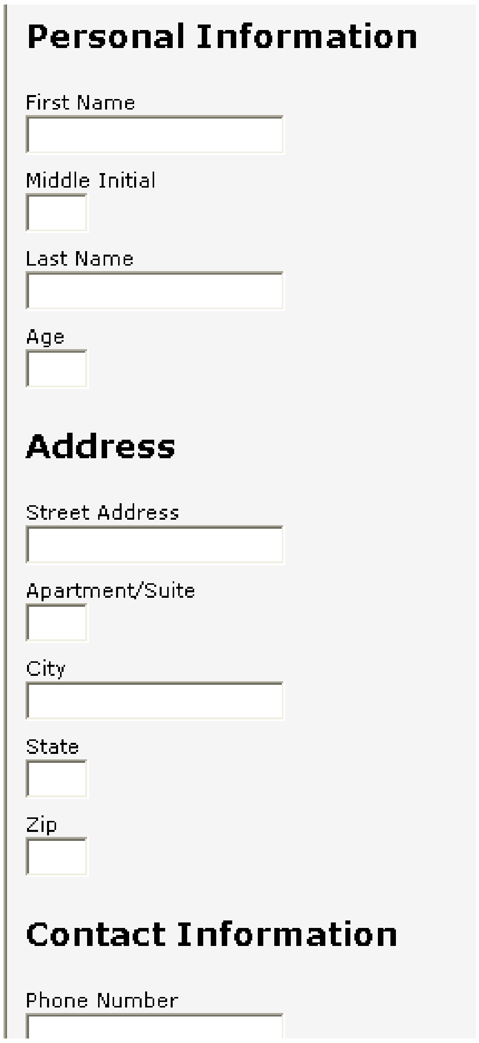
Multi Page Forms on One Page

Quite often you are faced with the prospect of a very long form. You could certainly break up the form into two, three or even four pages. However this has the consequences of having to somehow deal with the data that was filled out on previous pages. What we can do is use JavaScript to make the form appear as if we were moving through a series of steps (to keep the form manageable for user) while never leaving the page. Then we could just submit once and be done.

The Scenario

We'll assume we have a typical registration form. I've deliberately made the example form somewhat long but in reality such forms can still be awfully long and tedious. This is where this technique would come in handy.

Normally, you might have a form like this:


Personal Information

First Name

Middle Initial

Last Name

Age

Address

Street Address

Apartment/Suite

City

State

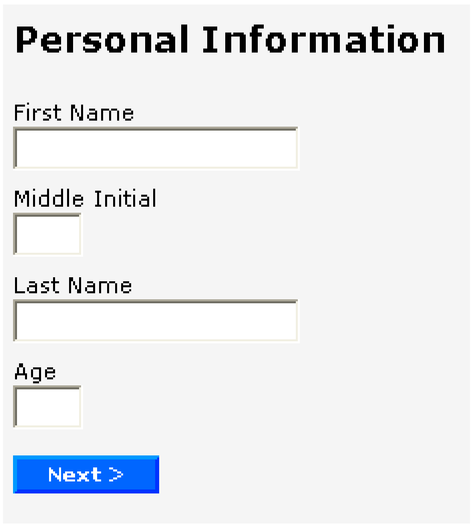
Zip

Contact Information

Phone Number

Notice the contact information doesn't even fit on the page. In reality such registration forms are even longer. Sure, we could make a table and give ourselves two columns to shorten up the form, but often that makes for a cluttered page. We'd like to have a nice clean interface for the user.

What we want is something that will prompt the user to the next section, like this:



Doing it this way we'll give the user nice short bites to digest, and then submit at the end when they are finished. We'll even provide a way for them to know at what stage they are in filling out the form.

The Form

The first thing we need is our form. In order to get the effect we want, we'll need to break up each section of the form that we wish to be its own page, and place it in a DIV tag. We'll do three steps in this case: Personal Information, Address Information and Contact Information, so we'll need 3 DIV tags:

```
<form name="registration">

<div id="personal">
<h3>Personal Information</h3>
<label for="first">First Name</label>
<input type="text" name="first" id="first" size="20" />

<label for="initial">Middle Initial</label>
<input type="text" name="initial" id="initial" size="2" />

<label for="last">Last Name</label>
<input type="text" name="last" id="last" size="20" />

<label for="age">Age</label>
<input type="text" name="age" id="age" size="2" />

<input type="button" value="Next &gt;" name="next" onclick="FormStep(2);" />
</div>

<div id="address">
<h3>Address</h3>
<label for="first">Street Address</label>
<input type="text" name="first" id="first" size="20" />
```

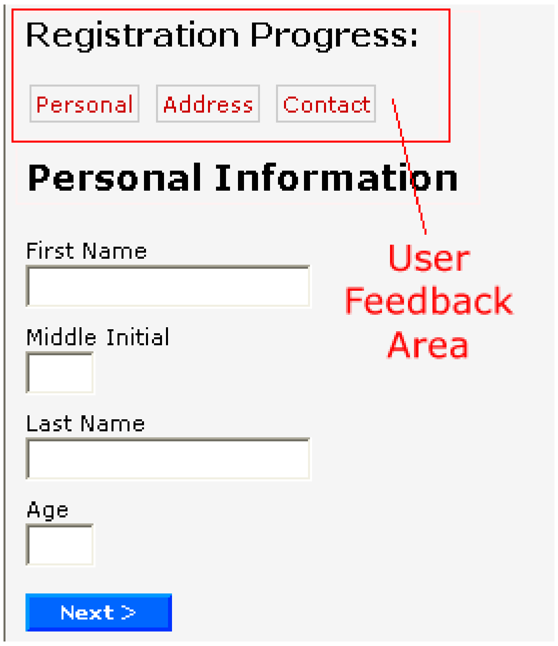
```
<label for="initial">Apartment/Suite</label>
<input type="text" name="initial" id="initial" size="2" />

<label for="last">City</label>
<input type="text" name="last" id="last" size="20" />

<label for="age">State</label>
<input type="text" name="age" id="age" size="2" />

<label for="age">Zip</label>
<input type="text" name="age" id="age" size="2" />

<input type="button" value="&lt; Back" name="back" onclick="FormStep(1);" />
<input type="button" value="Next &gt;" name="next" onclick="FormStep(3);" />
</div>

<div id="contact">
<h3>Contact Information</h3>
<label for="phone">Phone Number</label>
<input type="text" name="phone" id="phone" size="20" />

<label for="email">Email</label>
<input type="text" name="email" id="email" size="20" />

<input type="button" value="&lt; Back" name="back" onclick="FormStep(2);" />
<input type="submit" value="Register" name="submit" />
</div>

</form>
```

Each DIV tag was given a unique ID attribute, which we will use to hide and show the sections we need. We've placed next and back buttons in each section where appropriate, and we will use those buttons to call our function, which will be named **FormStep()**. In order to do that we must first set up the default version the user will see, which is only the personal information showing. So we'll place the following code in our CSS:

```
#personal { display: block; }
#address  { display: none; }
#contact  { display: none; }
```

Remember the **display** property controls whether or not to include the HTML code in the rendered page. Setting an element to **none** means it is effectively "removed" from the rendering of the page – it will not show up. Don't confuse this with the **visibility** property. Visibility will indeed hide an element, but it is not removed from the rendering code – so a blank space will be left where the element was. This gives us what we saw in the screen shot above.

Something else that might be nice would be to show the user at what stage they are at in filling out the form. Take a look at the following screen shot:



We'll change the progress blocks to green as they finish each section. The code for this section also uses some IDs and SPANs instead of DIVs. We use SPANs because they are an inline element (there are no line break) so they will all stay on one row:

```
<h4>Registration Progress:</h4>
<span id="per">Personal</span>
<span id="add">Address</span>
<span id="con">Contact</span>
```

We've styled them with the following CSS, which we will manipulate with our script:

```
#per { color: #cc0000; border: 1px solid #ccc; padding: 2px; margin: 2px; }
#add { color: #cc0000; border: 1px solid #ccc; padding: 2px; margin: 2px; }
#con { color: #cc0000; border: 1px solid #ccc; padding: 2px; margin: 2px; }
```

Setting Up the Script

To begin building the script, we'll need to tell it about our DIV elements that hold our form sections, as well as the SPAN elements that hold our "step" elements:

```
function FormStep(step)
{
     // DIVs holding each step of the form
     var personal = document.getElementById("personal");
     var address = document.getElementById("address");
     var contact = document.getElementById("contact");

     // SPANs to display progress
     var per = document.getElementById("per");
     var add = document.getElementById("add");
     var con = document.getElementById("con");
```

The script is going to be fairly straightforward and clean, which is always nice. We only need one parameter, and that is going to be the 'step" that the user is currently on. We'll adjust everything based on that. So when we call the function the first time to go to step 2, our button call looks like:

```
onclick="FormStep(2);"
```

Since we are going to step 2.

Which Step?

Somehow we need to take action based on the step number we have sent. Usually the initial thought is to start with some IF…ELSE control statements. While that would work, this is really the perfect situation to be using the SWITCH control structure.

```
switch(step)
{
     case 1:
     break;

     case 2:
     break;

     case 3:
     break;
}
```

Here we have the skeleton for the SWITCH. We're sending it the step number, and then based on which number we have, we'll enter that case – 1, 2 or 3. Note that "step" was the parameter sent, the name we gave it - FormStep(step).

Show/Hide the Right Step

In each case we'll have a different set of what shows and what doesn't. For example in the first case, we want personal information (1) to show, and address and contact information (2 and 3) to not show. We can accomplish this by changing the DISPLAY property of those three DIVs.

When we access this property, we must remember it is a property of the STYLE object related to the object you want to affect, so the format is:

object.style.display = block | none

So for step 1 we would have:

```
case 1:
personal.style.display = "block";
```

```
address.style.display = "none";
contact.style.display = "none";
break;
```

Now setting the other two steps is easy, step two will have address set as block, and step 3 will have contact set as block:

```
switch(step)
{
     case 1:
     personal.style.display = "block";
     address.style.display = "none";
     contact.style.display = "none";
     break;

     case 2:
     personal.style.display = "none";
     address.style.display = "block";
     contact.style.display = "none";
     break;

     case 3:
     personal.style.display = "none";
     address.style.display = "none";
     contact.style.display = "block";
     break;
}
```

Now, when each step is reached (whether by "back" or "next" button) the correct display will be shown.

Changing the Status Boxes

We want the status boxes up top to change along with the steps. The first thing we want to do is change the text itself from red to green. We already have our SWITCH statement all ready to go, so we merely need to put the code in the right block.

To change the text color we do the following:

```
per.style.color = "#c00";
```

Note we attach the **color** the same way we attached the **display** property. So then, our CASEs will now look like this:

```
switch(step)
{
     case 1:
     personal.style.display = "block";
     address.style.display = "none";
     contact.style.display = "none";

     per.style.color = "#c00";
     add.style.color = "#c00";
     con.style.color = "#c00";
     break;

     case 2:
     personal.style.display = "none";
     address.style.display = "block";
     contact.style.display = "none";

     per.style.color = "#060";
     add.style.color = "#c00";
     con.style.color = "#c00";
     break;

     case 3:
     personal.style.display = "none";
     address.style.display = "none";
     contact.style.display = "block";

     per.style.color = "#060";
     add.style.color = "#060";
     con.style.color = "#c00";
     break;
}
```

Notice that in step 1, all the colors are red, as they should be. In step 2 we set the first status box (personal information) to green. And in step 3 we set steps 1 and 2 to green. You'll notice that the third step really will never get set to green, since that is the last step they are working on.

What About Validation?

Validation for this type of form is going to have to be handled as though each step were a form. You don't want to be faced with the situation where the user submits the form and then there is an error in step 2 or 3. If the server sends the user back to the page, they are going to end up at step 1. What if the error is on step 2 or 3?

Therefore, you would want to validate any fields you needed to before allowing them to move on to the next step. If there are server-side validation considerations, where a JavaScript validation isn't possible (or desirable – such as password fields) then you could add some extra code and have the function called when the page loads. You would have to have the server side script tell the page what step the error was on. For example, if the error was on page 2, you would send a variable back to the page stating that as such. In php you could send it as a GET variable:

```
<?
$errorStep = $_GET['errorStep'];  // set by a server side script
if(!isset($errorStep)) $errorStep = 1;  // set it to one if there is no error
?>
```

Then insert that value in your onload call of the script:

```
window.onload = "FormStep(<?=$errorStep?>)";
```

I checked above to make sure my value was set, and if it wasn't, I set the value to 1 so the user would get the first step if there was no error (which is what you want at normal page load.)

Background Color on the Status

Lastly, we want to change the background color on the status fields to green as well. It's the same concept as changing the color:

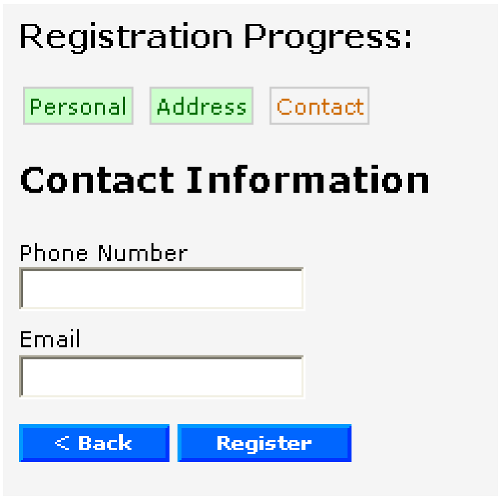
```
per.style.background = "#f5f5f5";
```

We'll place those lines right after the color setting, so the final SWITCH statement looks like:

```
switch(step)
{
     case 1:
     personal.style.display = "block";
     address.style.display = "none";
     contact.style.display = "none";

     per.style.color = "#c00";
     add.style.color = "#c00";
     con.style.color = "#c00";
     per.style.background = "#f5f5f5";
     add.style.background = "#f5f5f5";
     con.style.background = "#f5f5f5";
     break;

     case 2:
     personal.style.display = "none";
     address.style.display = "block";
     contact.style.display = "none";

     per.style.color = "#060";
     add.style.color = "#c00";
     con.style.color = "#c00";
     per.style.background = "#ccffcc";
```

```
        add.style.background = "#f5f5f5";
        con.style.background = "#f5f5f5";
        break;

        case 3:
        personal.style.display = "none";
        address.style.display = "none";
        contact.style.display = "block";

        per.style.color = "#060";
        add.style.color = "#060";
        con.style.color = "#c00";
        per.style.background = "#ccffcc";
        add.style.background = "#ccffcc";
        con.style.background = "#f5f5f5";
        break;
}
```

And the user would see something like this in the middle of the process:



The Full Script

The whole script looks like this:

```
function FormStep(step)
{
     var personal = document.getElementById("personal");
     var address = document.getElementById("address");
     var contact = document.getElementById("contact");

     var per = document.getElementById("per");
     var add = document.getElementById("add");
     var con = document.getElementById("con");

     switch(step)
     {
           case 1:
           personal.style.display = "block";
           address.style.display = "none";
           contact.style.display = "none";

           per.style.color = "#c00";
           add.style.color = "#c00";
           con.style.color = "#c00";
           per.style.background = "#f5f5f5";
           add.style.background = "#f5f5f5";
           con.style.background = "#f5f5f5";
           break;

           case 2:
           personal.style.display = "none";
           address.style.display = "block";
           contact.style.display = "none";

           per.style.color = "#060";
           add.style.color = "#c00";
           con.style.color = "#c00";
           per.style.background = "#ccffcc";
           add.style.background = "#f5f5f5";
           con.style.background = "#f5f5f5";
           break;

           case 3:
           personal.style.display = "none";
           address.style.display = "none";
           contact.style.display = "block";

           per.style.color = "#060";
           add.style.color = "#060";
           con.style.color = "#c00";
```

```
            per.style.background = "#ccffcc";
            add.style.background = "#ccffcc";
            con.style.background = "#f5f5f5";
            break;
      }
}
```

Conclusion

The display property can be a very useful tool in manipulating what the user sees based on certain conditions. The fact that it doesn't hide it, but actually removes it form the flow of the page is very useful. You could use the same concept for a permission based page or any number of conditional situations where you need to only show certain parts of a page.

Validating Forms: A JavaScript Validation Class

Let's face it, validating forms is a necessary evil. Writing good validation functions can be tedious, time consuming and if you don't do it right you could be in a world of pain. There are numerous reasons why you have to validate form fields. Sometimes you need alphanumeric characters, sometimes only alphabetic. Maybe you need a number or a certain kind of number, like a telephone number. Not to mention email addresses, social security numbers and the behemoth of all validations, the credit card.

Wouldn't it be really useful to have a JavaScript class (an object) with validation methods that you could reuse again and again? That was the purpose of my chapter on JavaScript objects. Since we have a basic understanding of objects, we can now move on and build a real validation class so you aren't constantly doing the same work over and over.

Validation: Have a Strategy

We would be remiss without mentioning that client side validation is **not enough**. Just like your security strategy for your PC is much more than a simple anti-virus application, your validation strategy is more than client side JavaScript. The server must also take care to make sure it is working on valid data. This is especially important with e-commerce systems and web applications that handle lots and lots of data.

What you are doing by validating your data on the client side is making the whole validation strategy more bulletproof and safe for the application at hand, as well as making the process smoother and faster by trapping as many errors as you can in the browser before sending it over the Web to the server.

A Validation Library

It would be nice if you had a library of functions to validate your form fields. Once you build a library, you could then even build a validation class if you were so inclined to do so. But to start, just having some good validation functions can be worth its weight in gold.

We will introduce briefly one advanced concept for validation and that is that of Regular Expressions or RegEx for short. Regular expressions can get extremely complex and are therefore very powerful tools. While it is beyond the scope of this one paragraph (or many paragraphs) to teach you RegEx, we'll at least introduce some simple patterns to get your feet wet and show you how effective it can be in validating fields.

What is a Regular Expression?

Regular expressions give you a means to match patterns. That's about as simple as you can put it. So if I had a string that contained this:

```
var myString = "billybobjoelarry";
```

I could use a regular expression to see if the pattern "bob" was somewhere inside (which it is.) Pattern matching is powerful because you can use it to find out things that you either cannot do with plain string methods in JavaScript (such as **substr()**), or it would be extremely difficult to do so.

In validation, you need to find out things like:

- Are there numbers where there should only be letters?
- Are there letters where there should only be numbers?
- Is this telephone number the right format?
- Is this postal code the right format?

Pattern matching is the most powerful tool you can use to do these kinds of operations. We won't use RegEx for everything, but eventually you could.

How We'll Build It

What we're going to do is build a main validation function that automatically runs the right "sub" function for each field. We'll tell the main function which sub function to run by including the sub function name in the element ID and extracting it.

The idea is to be able to have an abstract function that can be placed in any file. Then what you actually put in the tags determines the validation. You won't have to necessarily edit the main function all the time, which is the part that can become really tedious.

A Word About Validation Techniques

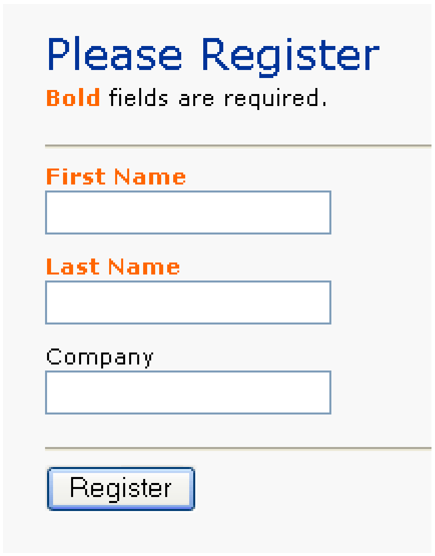
Remember that there are lots and lots of ways to validate fields, and to let the user know that something is wrong. In this case, we'll halt the validation as soon as an error occurs and let the user know. As always, a well-designed form with good visual cues on what the user should do will help limit errors quite a bit.

Empty Fields

One of the most common validations is empty fields. If the user misses a field, the server must send the page back, with all the data they already filled in saved. We'd like to prevent this from happening before the page gets sent off. Validating this is made tricky because the user could inadvertently (we hope) add a single space to a form field. That means that checking for null and an empty string won't do the trick. So to start we are going to have three functions: the main validation function, the **isEmpty** function and a function strip whitespace off of a string.

The Form

First we have a simple form that asks for a user's first and last name, along with company name. Both name fields are clearly marked as required:



And the HTML:

```
<form name="info" onsubmit="return Validate(this);">

<label for="first_isEmpty" class="required">First Name <input type="text"
name="first" id="first_isEmpty" value="" /></label>

<label for="last_isEmpty" class="required">Last Name <input type="text"
name="last" id="last_isEmpty" value="" /></label>

<label for="company">Company <input type="text" name="company" id="company"
value="" /></label>
<hr />
<input type="submit" name="submit" value="Register" />
</form>
```

A few things to notice. First of all, as I said the **ID** will determine the validation used:

```
id="first_isEmpty"
```

So you put your field name, and underscore and then the validation. You can see that having simply this to do for your validation, and not modifying the code each time, is a nice way to validate your pages.

We'll use the **onsubmit** event of the form, with the **return** keyword to control the form submission. Remember this is the correct way to validate a form in most cases. This allows you to return **false** from your validation to prevent form submission. The onclick handler of the submit button does NOT give you this option!

```
onsubmit="return Validate(this);"
```

Also notice that the form uses the **this** keyword to pass itself to the validation logic.

The isEmpty() Function

Let's first take a look at the functions that will validate that our field is not empty.

```
function isEmpty(eString, fieldName)
{
     eString = trimWhitespace(eString);
     // make sure the field contains some kind of data.
     if(eString == null || eString == "")
     {
           alert(fieldName + "must not be blank.");
           isValid = false;
     }
     else
     {
           isValid = true;
     }
}
```

The two parameters we want to pass are: 1) the string that we are validating (which is the value of our current text field) and 2) the name of the field (meaning the actual user friendly text name, such as "First Name"). We'll test the string to make sure it isn't empty, and we'll use the field name to tell the user which field has the problem if it is.

The first thing you will notice is that we immediately call another function on our string first:

```
eString = trimWhitespace(eString);
```

Remember, a single space will be considered "not empty" when checking for null or an empty string. So we need to strip any spaces, and that is what the following function will do:

```
function trimWhitespace(strText)
{
     // remove leading spaces
while (strText.substring(0,1) == ' ')
{
        strText = strText.substring(1, strText.length);
     }

     // remove trailing spaces
while (strText.substring(strText.length-1, strText.length) == ' ')
{
        strText = strText.substring(0, strText.length-1);
     }

   return strText;
}
```

This nifty little function is going to take that string, which again is the current value of our text field, and remove any whitespace. It removes that space from the front and back the same way, using the **while** condition. This condition says "as long as". So in this case "as long as there are characters in my string" then do what is inside the braces.

```
while (strText.substring(0,1) == ' ')
{
        strText = strText.substring(1, strText.length);
     }
```

The condition tested on here uses the **substring()** method. The syntax for using it is very straightforward:

```
object.substring(indexA, indexB)
```

You tell the method to look at the characters including and between the two indexes. So if I had a variable called **theName** with the string "bob" in it and did this:

```
theName.substring(0,1);
```

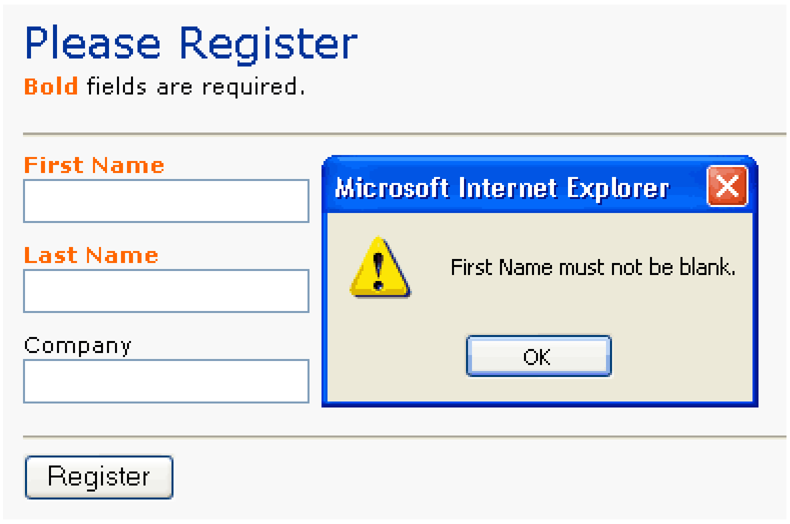
The result would be "bo". In the while condition above, the test is made on the first character to see if it is indeed a blank space (note the space between ' and '). As long as that is true, the empty space is removed by resetting the strText variable to a new substring WITHOUT the first character – note how it starts with a 1, not a zero. It can do this as long as there are whitespaces. So if there are 5 whitespaces, it will strip all 5.

The second part does the near exact same thing, only acting on the last character of the string instead. With this nifty trick in place, we can finish looking at the isEmpty function.

```
if(eString == null || eString == "")
     {
           alert(fieldName + "must not be blank.");
           return false;
     }
     else
     {
           return true;
     }
```

Now that we have a string without whitespace (which just might now be empty) we test to see if it is either null or empty (""). Both may not be necessary, but it doesn't hurt to have them. The double || signifies logical OR, meaning if either condition is met, it will evaluate to true and enter the braces.

If the string is blank, we alert the user that it cannot be and even tell them which field is causing the trouble:


Please Register

Bold fields are required.

First Name

Last Name

Company

Microsoft Internet Explorer

First Name must not be blank.

OK

Register

We then set a variable called **isValid** to false. If the string is not empty, we set that value to true. At the end of our validation function, if isValid is false the form will not submit, and conversely, if it's **true** everything is fine and we'll be on our way.

Now let's look at the heart of the matter – the validation function itself:

```
var isValid;

function Validate(oForm)
{
	// set up error flag
	isValid = true;

	// iterate through all form elements
	for(var i=0; i<oForm.length; i++)
	{
		// get current element
		var currElement = document.forms['info'].elements[i];
		var currID = currElement.id;
		var currName = currElement.name;

		// get validation method (if any)
		var val = currID.split("_");
		var valMethod = val[1];
		var fieldName = currElement.parentNode.childNodes[0].nodeValue;

		// check for an run any validations
		switch(valMethod)
		{
			case "isEmpty":
			var eString = currElement.value;
```

```
                  isEmpty(eString, fieldName);
                  break;
            }
      }
      return isValid;
}
```

There is quite a bit going on here so we will take it one step at a time. I should mention that we are using DOM methods here which allow this script to work in DOM compliant modern browsers, and this was tested in IE6, NS7, Mozilla and Opera7.

First we are receiving the form as a parameter (**oForm**) which we sent using the **onsubmit** event handler. Having our form object present, we want to loop through the entire form, looking for elements that require validation. Remember we tagged such elements with a "_isEmpty" after the name for the field. But notice as we go through this function that there are no local, specific ties to the document – it is abstract from the form which makes it very flexible.

So first we iterate through each form element with the FOR loop:

```
for(var i=0; i<oForm.length; i++)
```

Here I put the length property attached to the form object right inside the FOR condition instead of setting up a variable for it beforehand. This is perfectly acceptable, and once you get more used to scripting, it's a nice way to save space.

Next, we get the current form element in the loop we are dealing with:

```
// get current element
var currElement = document.forms['info'].elements[i];
var currID = currElement.id;
var currName = currElement.name;
```

We are now inside the **FOR** loop. Using the DOM level 0 method of referencing forms (the forms and elements arrays) we can easily grab the element that is in indexed position. In the first iteration, it's the "first name" field, then it will be "last name", then "company", and lastly it will be the submit button.

Using the ID property, we find out what the ID of that element is – remember this is where we placed our validation flag. So how do we find out if the element wants a validation – and which one?

```
var val = currID.split("_");
var valMethod = val[1];
var fieldName = currElement.parentNode.childNodes[0].nodeValue;
```

Since we have the ID as a string, we can run a little method on it called **split()**. This guy takes a delimiter you supply, and returns an array with the delimiter removed. If you gave it a comma separated list like this: "1,2,3" and split it on the comma, it would give you an array:

1

2
3

So this is exactly what we need. We split on the current ID (which is the first case is first_isEmpty). This gives us:

First
isEmpty

as an array. In the second line of the snippet above, we then grab that second value **-val[1]**- , which just happens to be "isEmpty"! Viola – our validation call! We place that in our **valMethod** variable (short for validation method). Now we just need to get the name of our field. This involves a bit of DOM trickery. It would be great to use the innerText property here, but Netscape doesn't support that.

But we do have our current element as **currElement**. Using the **parentNode** method, we can get the **LABEL** tag it is nested in. The parent node is the node that contains the element we are currently on. Since the **INPUT** is nested in the LABEL, the LABEL is the parent. Now the label has, you guessed it – the text we need. The only way to grab that text in Netscape is to use the the methods **childNodes** with an index and then get that node's value using **nodeValue**.

Since we know the text is the first node of the label, we we specify that as **childNodes[0]**. Then we simply attach nodeValue to grab the text. Notice how its all one long string – you just keep piling on the methods until you get what you need. As long as the methods are valid, the string can be as long as it needs to be.

Now we have all the information we need for the final code.

```
switch(valMethod)
{
     case "isEmpty":
     var eString = currElement.value;
     isEmpty(eString, fieldName);
     break;
}
```

Even though we have only one case right now, we're going to extend this, so using **SWITCH** makes a lot of sense here. We'll be able to add as many functions as we want. We know what the validation method should be for our current element, so we look for that in the SWITCH. The first case matches in our example of course, so we enter inside. We set a variable to hold the current element's value (somebody's first name) and send that to the isEmpty() function which we explained above.

Remember that inside that function our isValid variable is set to true or false, depending on whether it passed the blank/no whitespace test. The last thing we need to do is return that variable:

```
return isValid;
```

At this point the **isValid** variable is false if there has been an error, so the form will not submit. Otherwise it is true and the form submits.

Conclusion

We've only scratched the surface of how we can use this validation technique. We're going to add some more validations to the function and see how it works. We'll use those scary regular expressions in a couple too! The goal here is to have a validation function that doesn't have to be touched over and over again each time you start a new project, but it can be extended or customized at will. Add in the validation functions you need and leave out the ones you don't.

Email address and password validation

In the previous paragraph we built a base script to function as kind of a gateway for our library function. This function allows us to add validation function names to our form inputs, read them and run the correct validation. It then returns the user back to the form if an error was made. We're going to add a new validation function to our form. We'll even use a simple regular expression to make the validation as well.

To review, here is the main validation function:

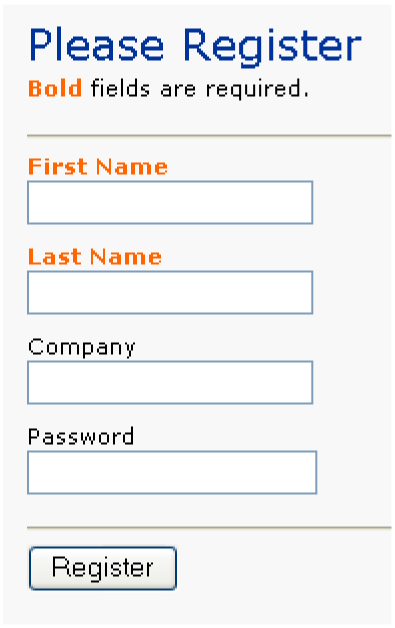
```
var isValid;
function Validate(oForm)
{
     isValid = true;
     // iterate through all form elements
     for(var i=0; i<oForm.length; i++)
     {
           // get the validation method name from the element ID
           var currElement = document.forms['info'].elements[i];
           var currID = currElement.id;
           var val = currID.split("_");
           var valMethod = val[1];

           // get the field label name
           var fieldName = currElement.parentNode.childNodes[0].nodeValue;

           // call the correct method
           switch(valMethod)
           {
                 case "isEmpty":
                 var eString = currElement.value;
                 isEmpty(eString, fieldName);
                 break;
           }
     }
     return isValid;
}
```

The **SWITCH** control allows us to add each new library function as needed. We wrote a function that checked for an empty field. We will add a function to make sure that the correct data has been entered.

The Form

Once again we have a small registration form. Let's add another required field that we have to validate, a password field. Generally for passwords you only want to allow letters and numbers, but no special characters and no spaces. This is the type of thing for which regular expressions work really well. Remember, we're not going to teach regular expressions in this series, that isn't the intent. We'll give you just enough to whet your appetite. If you want to get started on your own, there is a fairly good introduction at javascript kit.

Here is our expanded form:



We've added the password field. Below is the full HTML for the form:

```
<form name="info" onsubmit="return Validate(this);">

<label for="first_isEmpty" class="required">First Name <input type="text"
name="first" id="first_isEmpty" value="" /></label>

<label for="last_isEmpty" class="required">Last Name <input type="text"
name="last" id="last_isEmpty" value="" /></label>

<label for="company">Company <input type="text" name="company" id="company"
value="" /></label>

<label for="pw_isPassword">Password <input type="password" name="regPassword"
id="pw_isPassword" value="" /></label>
<hr />
<input type="submit" name="submit" value="Register" />
</form>
```

Notice again that we added the method name we are going to use to validate the field to the ID of the form element, preceded by an underscore (in bold in the example above for clarity). So we need to add that method to our library.

isPassword Method

In addition to making sure that the password contains only legal characters, we should also check to make sure that it isn't left blank. And we might also want to make sure that it contains 6 or more characters, since longer passwords are better.

Does this mean we need extra functions for those? And we already have an **isEmpty** function. How are we going to call two functions form one form element? Adding a second underscore and checking against that would get awfully messy!

Luckily, we don't have to do any such thing. All these functions (which we are also calling *methods* for our main validation function) that we are writing are self contained. Remember last time we called the **trimWhitespace()** function from *within* the isEmpty function. Well they can *all* be used in a similar fashion. Once you have the validation process running, there is plenty of flexibility available to take care of the validation you need to have done.

To show this, take a look at the new **isPassword** method:

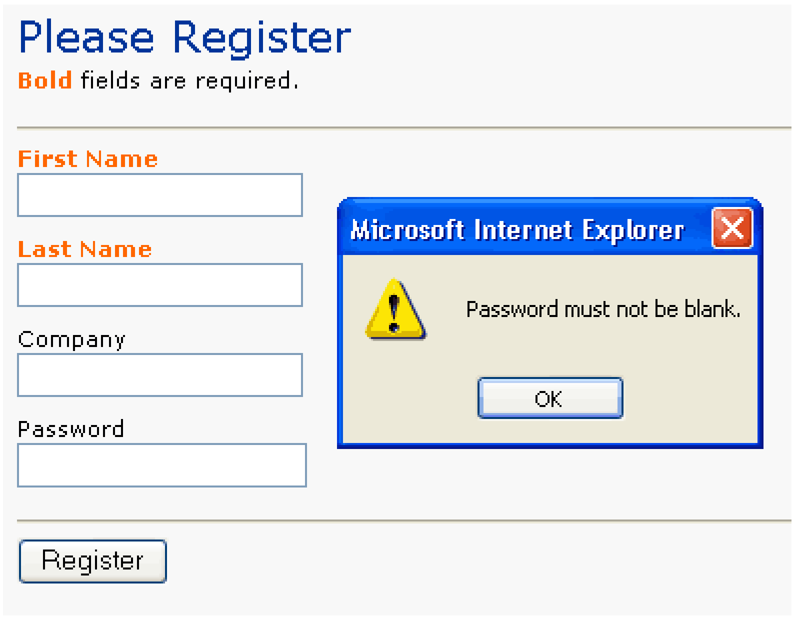
```
function isPassword(eString, fieldName)
{
     isEmpty(eString, fieldName);

     var pwRegEx = /[^a-z\d]/i;
     var passwordValidates = !(pwRegEx.test(eString));

     if(passwordValidates == false)
     {
           alert(fieldName + "must only contain alphanumeric characters.");
           isValid = false;
     }
     else
     {
           isValid = true;
     }

     if(eString.length < 6)
     {
           alert("Password must be a minimum of 6 characters long.");
            isValid = false;
     }
}
```

You'll notice that we have sent the same parameters to this new method as we did to isEmpty: the string contained in the form element and the field name of the form element (in this case "Password".) Remember that this is being called in a loop in our validation function via the SWITCH statement, so we need to add our new method to that:

```
switch(valMethod)
{
     case "isEmpty":
     var eString = currElement.value;
     isEmpty(eString, fieldName);
     break;

     case "isPassword":
     var eString = currElement.value;
     isPassword(eString, fieldName);
     break;
}
```

Now when "isPassword" is attached to a field ID, the isPassword method is called. And what do you know, the first thing we do is call the **isEmpty** function! After all, the password cannot be empty either. All we need do is pass along the parameters we already have, and the function works just like last time.

If I go ahead and leave the password blank then, I will get the warning from the isEmpty function:


Please Register

Bold fields are required.

Microsoft Internet Explorer

Password must not be blank.

Very nice, that is now taken care of. But now comes some regular expression nastiness!

```
var pwRegEx = /[^a-z\d]/i;
var passwordValidates = !(pwRegEx.test(eString));
```

Regular expressions can be awfully intimidating. I know they still are for me, especially when they get very complex. But the root of how they work is not really that difficult. What you are doing is setting a variable (**pwRegEx**) to a regular expression pattern. This pattern will look for matches in a string. In this case our pattern is the following:

```
/[^a-z\d]/i
```

The "/" characters specify the beginning and ending of the pattern, while the "i" signifies "ignore case" so it is case-insensitive. Furthermore the caret (^) means "not" (because we want to match things in our string that do NOT belong there!) So the real pattern is:

```
[a-z\d]
```

And all this is saying is "any alpha character a through z, and any digit." The "\d" is "any digit" and of course "a-z" is literally a-z. So anything that is NOT one of those characters is what we are looking for.

To do this, we'll use the **test()** method. This method provides the best way to see if your pattern finds a match in the string you send. We want to make sure that there aren't any characters that are NOT alphanumeric in our string. Now this part can get tricky because it seems like we have two negatives here (and doesn't that make a positive?)

- First, we have a NOT in our regular expression (^) that says match something that is NOT an alphanumeric character. That is our test.

- Second, we have the test method being called on our string that also has the NOT operator (!) in JavaScript. Why? What test() is going to return is either a true or false value. If it finds a match to the pattern, it returns true. Sounds logical right? And vice versa, if no match, it returns false.

Take this test string: "453&3". It has an invalid character – the "&". If we send that to the test() method with our regular expression, we are saying this:

"Please match any the characters in my string (453&3) that are NOT alphanumeric." Well, one character isn't – the & sign. So we have a match correct? Then test() is going to return TRUE. See the problem? Logically we want a FALSE value to show the error. I suppose we could change our IF condition to test for a true result, but that doesn't make logical sense, and that's never a good idea to program that way. So we add the NOT operator before the test() call. That way if there is a match, we'll get a FALSE return.

This works because the test() still does the same match, and still finds it. But because the NOT operator is lurking outside the parenthesis:

```
!(pwRegEx.test(eString))
```

You see, we DO have a match here, remember we had an illegal character. But the NOT operator changes it to say "check and see if we DON'T have a match" and so the value returned is FALSE. This can be a bit tricky to wrap your head around, but re-read it if you are having trouble with it and play with removing the NOT operator yourself and see what happens. Practical testing is a great way to learn.

Setting isValid

Now just like before we have to set the global **isValid** variable to prevent form submission if there is an error. We also, of course, need to tell the user what is wrong.

```
if(passwordValidates == false)
{
     alert(fieldName + "must only contain alphanumeric characters.");
     isValid = false;
}
else
{
     isValid = true;
}
```

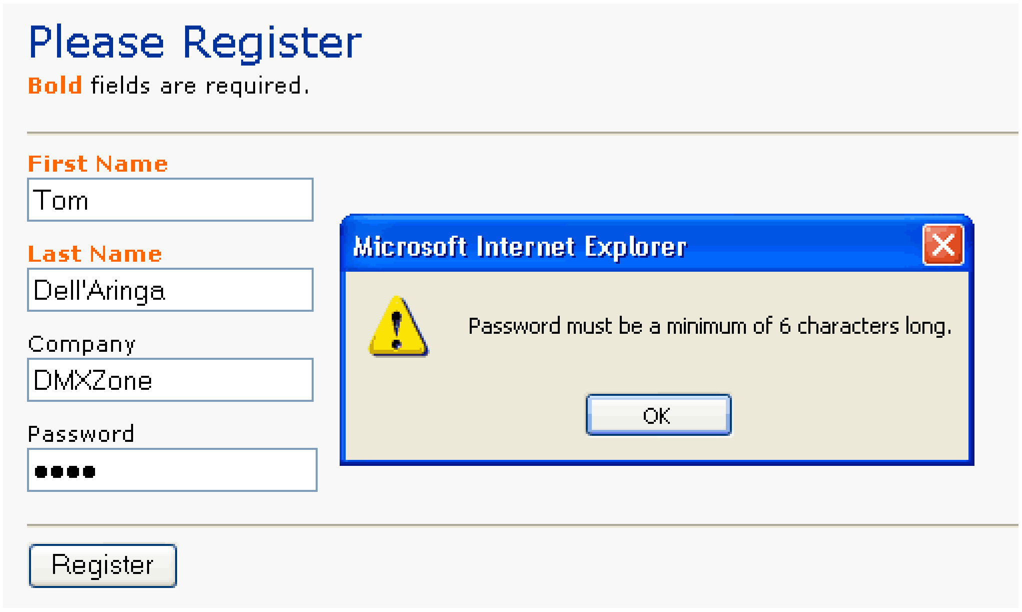
So it turns out that if we get a false value, it means the password did not validate, there was a match of a bad character. So we tell the user so, again using the field name so they know what field they made the mistake in. Then isValid is set to false, the form will not submit.

Otherwise, things are fine and isValid is true.

Lastly, we wanted to check to make sure the string was at least six characters long.

```
if(eString.length < 6)
{
     alert("Password must be a minimum of 6 characters long.");
     isValid = false;
}
```

That's easy enough to do by simply checking the length property of the string, making sure it is not less than six. So even if they do everything else right but submit a 4 character password, its still an error:


Please Register
Bold fields are required.

First Name
Tom

Last Name
Dell'Aringa

Company
DMXZone

Password
●●●●

Microsoft Internet Explorer
Password must be a minimum of 6 characters long.

Note that this doesn't actually test that the password entered matches that of the user; that needs to be done on the server, against the master user/password database. What we've done is trap invalidly formatted passwords before they even get sent to the server, reducing wait time, presses of the submit button, and server processing.

isEmail – Validate an Email Address

So let's add one more, the dreaded email address. Email addresses are very difficult to validate. Not only are there many legitimate ways to write an address, but just because the email address is in a valid form it does not mean it's a real working address. But at least making sure it is some kind of valid form will certainly help.

What are valid formats?

name@domain.com
name@domain.info
name@domain.ca
name@domain.co.uk or org.uk
name@domain.win.net
name.name@domain.com
name@subdomain.domain.com
name.name@subdomain.domain.com

That's a nasty list to deal with and probably doesn't include quite everything. But there is a regular expression that can handle this.

Here is our function:

```
function isEmail(eString, fieldName)
{
     var emailRegEx = /^w(\.?\w)*@\w(\.?[-\w])*\.[a-z]{2,4}$/i;
     var emailValidates = !(emailRegEx.test(eString));

     if(emailValidates == false)
     {
           alert(fieldName + "must be a correctly formatted and valid email
address.");
           isValid = false;
     }
     else
     {
           isValid = true;
     }
}
```

The regular expression looks daunting as usual. I won't go into an explanation of how this works, but it does work!

From there we follow the exact same pattern for the password function. Use the test() method to find out if a pattern is matched, with the NOT operator surrounding it. Again we can check for a false value and alert the user accordingly.

Again you need to add the form field to your form:

```
<label for="email_isEmail" class="required">Email <input type="text"
name="email" id="email_isEmail" value="" /></label>
```

Noting the "**_isEmail**" tacked on to the ID. And then we add that method to our SWITCH:

```
case "isEmail":
var eString = currElement.value;
isEmail(eString, fieldName);
break;
```

Variable Scope Problem

Now if you have really been paying attention, or you've really been testing the form you have realized we have a problem. If we have an error in the password field, and no error in the email field – the form submits. Why? We got the alert, and isValid was set to false wasn't it?

Yes, it was. However it was RESET to true when the email validated as true. Now we have a variable scope problem. We're resetting our isValid variable too often. What we have to do is actually fairly simple but it will change how things work. Instead of getting string of ALL errors in alert boxes, the user will get the FIRST error and then be returned to the form.

In my view, this is maybe even a better way. Clicking through a bunch of alerts can be a pain, and often once you realize you made a mistake you can change other mistakes on your own. Maybe you submitted the form oby accident. Generally a user only makes one or two errors on a form anyway.

What we need to do is test the isValid variable while we are still looping through all the form elements. If isValid is *ever* false, we RETURN false right there and halt form submission.

Once we make it safely out of the FOR loop, we can RETURN true and be finished. Here is the modified validation function:

```
var isValid;
function Validate(oForm)
{
     isValid = true;
     // iterate through all form elements
     for(var i=0; i<oForm.length; i++)
     {
           // get the validation method name from the element ID
           var currElement = document.forms['info'].elements[i];
           var currID = currElement.id;
           var val = currID.split("_");
           var valMethod = val[1];

           // get the field label name
           var fieldName = currElement.parentNode.childNodes[0].nodeValue;
```

```
            // call the correct method
            switch(valMethod)
            {
                  case "isEmpty":
                  var eString = currElement.value;
                  isEmpty(eString, fieldName);
                  break;

                  case "isPassword":
                  var eString = currElement.value;
                  isPassword(eString, fieldName);
                  break;

                  case "isEmail":
                  var eString = currElement.value;
                  isEmail(eString, fieldName);
                  break;
            }
            if(isValid == false) return false;
      }
      return true;
}
```

Notice the bold bits at the end, where we made the change.

Conclusion

You can keep adding more validation methods to work with your main function until you have everything covered for your form.

In the case of the email, if that should not be blank you can simply add the call to the isEmpty() method inside the isEmail() method. Once you have this library written, you should not have to keep rewriting everything over and over. Some things will always need tweaking it seems, but the majority of the work will be there for you to start with. This can be a big time saver.

Formatting User Form Data

Often when you have a form, the user inputs data that either doesn't match the format you desire, or it isn't as nice looking as it could be. A good example is the phone number. The typical 7 digit phone number looks like 555-1212. Quite often, people entering phone numbers on web forms omit the dashes and enter 5551212 instead. While there isn't anything wrong with this, there is a chance that an error is harder to spot by the user.

What you could do is check the value of that phone number after it has been entered. If the dashes have been left out, you could simply add them. You could do the same thing for a 10 digit phone number or even an international phone number. Of course the more data you handle the more difficult it's going to be.

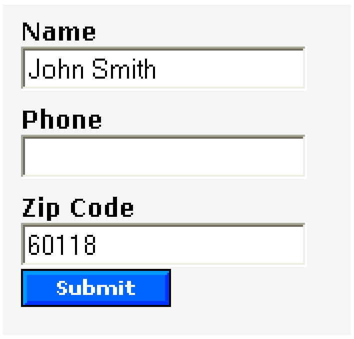
In this paragraph we'll take a look at 7 and 10 digit phone numbers and how we can format (or mask) them to look nicer should the user decide not to enter dashes. While we're doing this we can introduce some handy string methods and talk about a very basic regular expression.

A foundation

As I have done in the past, I'm going to give you a script that is a building block for further functionality. While it will certainly work on its own as far as our desired functionality requires, it can be extended to be part of a larger application.

For example, we won't cover validation of the phone number in this script. We've talked about validation in the past, and certainly you will want to validate a phone number if it is an important part of your application. We will point out where you would want to do such validation however.

So as usual, we have a simple form that we'll deal with. In this case we'll assume we have a small portion of some kind of application that needs personal information. We have the following:


Name
John Smith
Phone

Zip Code
60118

This simple form has the following HTML:

```
<form name="register">
<label for="name">Name</label>
<input type="text" name="name" id="name" value="John Smith" /><br />
```

```
<label for="phone">Phone</label>
<input type="text" name="phone" id="phone" onblur="FormatPhoneNumber(this);"
/><br />
<label for="zip">Zip Code</label>
<input type="text" name="zip" id="zip" value="60118" /><br />
<input type="submit" value="Submit />
</form>
```

You'll notice in bold the event handler **ONBLUR** which we have not talked about yet. We're going to use this as the trigger to run our script.

What's a blur and why is it on?

Good question. **Blur** happens to be the opposite of **focus**. When something comes into focus you have your attention on it. When something goes out of your focus, you are no longer paying any attention to it, it's "blurred" in your attention. So the idea of onblur is that when the focus leaves the object (that is, when you tab or click away from the item) the event handler fires.

This is perfect for our needs, because we want the phone number to be formatted as soon as the user moves away from the text input. Most people take one look, if even a quick one, before they submit a form. If the phone number is nice and formatted they can easily scan to see if it is their correct number.

How the script will work

So how should our format or mask script work on the data? We'll accept 7 and 10 number phone numbers, and we'll even go so far as to put brackets around the area code should they provide it. A dash will separate typical 3 and 4 number break.

What we will not do is validate the phone number here. I've found in the past that doing validation on the onblur event can be a bit troublesome. There's a chance you could end up in an odd loop of errors because the user cannot understand they need to re-focus the input where the error is to fix the problem.

The best thing to do is handle validation during the form submission. Hopefully if they do enter an invalid phone number, say 6 numbers or 11, they will realize that fact when it gets reformatted to a long string of numbers with no dashes or brackets.

Manipulating strings: The substring() method

In order to do what we want, we're going to need to know where to put dashes and brackets in our phone number string. Fortunately, JavaScript provides two different methods for picking apart strings. They can be easy to confuse (in fact, I still do it all the time) because they are so closely named. The one we will be using is substring() and the other is substr() – (see what I mean!)

The substring() method returns the characters in a string between two specified indices as a substring (thus the name.)

Syntax

```
object.substring(indexA, indexB)
```

You provide the two arguments, indexA and indexB as numbers. For example, if we wanted the word "pot" out of "spots" which was placed in a variable called someVar, we could do this:

```
var newString = someVar.substring(1,4);
```

Starting at 0 of course, the "p" falls in the 1 position, so we take that letter. For the ending position, you have to be careful. The indexB position is the position AFTER the last character you want. If I had done 1,3 we would have only gotten "po". With this knowledge in hand we can begin the script.

Setting up

First things first, let's get our data from the input field:

```
function FormatPhoneNumber(oPhone)
{
var strNewPhone;
var strPhone = oPhone.value;
}
```

The variable **strNewPhone** is going to hold the new, formatted string that we are going to send back to the form. So we initialise that right away. You'll notice we sent the object itself (the text field) using the **this** keyword with our onblur event, above. That means we have the object as our parameter, **oPhone**.

Remember though, oPhone does not hold the actual phone number, but the object itself. Rest easy however, all we need do is grab the value property of the text field, and viola, we have our phone number just as the user entered it.

Trouble on the second ride

Here's where we run into a problem. If the user goes back and makes a correction to the number, the value for our phone number is going to hold possibly spaces, dashes and brackets. Yet, we're going to add these things in just a moment. That means we can end up with a horrible mess of a phone number.
What we have to do is "clean" the phone number each time we run the script. We need to remove all three of those potentially phone number damaging items: spaces, dashes and brackets. This is where regular expressions come in!

Don't fall over just yet!

I know what you're thinking: "regular expressions are *nasty*." They certainly can be. However, if you can get at all proficient enough to understand the basics, you'll find out just how powerful they are. Then you can pick up more as you go along. (Well, that's my approach anyway!)

But what is a regular expression? In its most simple terms, a regular expression is a *pattern*. You have some kind of pattern you want to match. Regular expressions provide the tools to describe that pattern and test it. Furthermore, there are other methods that allow you to take that expression, execute it on a string and get back the result. So you can remove characters, add them, and do all sorts of contortionist tricks on them. Sound intimidating? Again, it can be. But we're only going to use 3 pretty simple cases to get your feet wet.

RegEx syntax

There are all kinds of ways to build these patterns. I'm not going to go into a regex tutorial here, I'm only going to provide some tidbits. If you are a glutton for severe punishment… I mean if you want more information, take a look at http://www.js-examples.com/javascript/core_js15/regexp.php3.

Okay, we need to remove dashes, let me drop a function on you, then I'll explain it.

```
function StripDashes(strValue)
{
     var objRegExp = /-/g;
     return strValue.replace(objRegExp,'');
}
```

Eeew, gnarly! (My editor will love that one.) Okay what is going on here? Honestly, not much. The first thing to know is that we begin and end our pattern with the backslash character "/". So what is inside those two slashes is the pattern. Looking inside there we see… one lonely dash! That's our pattern! After all, we want to remove dashes. That's almost too simple. But wait you say, what's the "g" afterward? Well that little guy stands for "global." In other words, look throughout the WHOLE string for such values.

Okay, we've taken that pattern and placed it in a variable. Now regex knows what to look for. But we need more than that, we need to replace each occurrence of the dash with something else – in this case, nothing – we want to remove it.

That's where the **replace()** method comes in.

Replace()

JavaScript provides this method free of charge to use with regex. What it does is alarmingly simple and powerful. It takes a regular expression as the first argument (**objRegExp** in our case) and what you want to replace each occurrence of the pattern (each match) with (nothing in our case, an empty string or ''.) That's somewhat oversimplified in a sense (and purists will argue with much of this) but the point is we want to learn how to use these things first, not regular expression theory.

Now we want the new string, with all the matched patterns replaced with empty strings, returned back to us. So we just use the **return** keyword, which tells the function to throw back something to the caller (whatever it was that called the function). Normally you've seen us do return **true** or return **false** in these paragraphs. Well we can return any value we want. In this case, we're going to return the modified string we just ran the regex on.

```
return strValue.replace(objRegExp,'');
```

This is a shorter version of doing something like:

```
var fixedString =  strValue.replace(objRegExp,'');
return fixedString;
```

The extra step of declaring a variable, setting the result as the new value and returning it is really unnecessary. The shorter version just says "return the value of what this method evaluates to." The **strValue** here is the phone number we are running the script on – we'll look at that now.

Calling the regex function

Seems simple enough, we just need to call the function. All we need do is pass our user-entered phone number to the function, and get back a corrected string. Simple:

```
function FormatPhoneNumber(oPhone)
{
var strNewPhone;
var strPhone = oPhone.value;
     strPhone = StripDashes(strPhone);
}
```

Simple, but powerful - strPhone now holds the string the user entered as their phone number minus any dashes! Now stripping the spaces is just as easy:

```
function StripSpaces(strValue)
{
     var objRegExp = / /g; //search for spaces globally
     //replace all matches with empty strings
     return strValue.replace(objRegExp,'');
}
```

```
function FormatPhoneNumber(oPhone)
{
     var strNewPhone;
     var strPhone = oPhone.value;
     strPhone = StripDashes(strPhone);
     strPhone = StripSpaces(strPhone);
}
```

Notice the new pattern for spaces above – it's a space! Now that's easy. It works exactly the same way. Now we have a string with no dashes or spaces. We're almost ready. The last thing we need is a pattern to clear any brackets, should they exist. This regex is a bit more complex:

```
function StripBrackets(strValue)
{
     var objRegExp = /[\(\)]/g; //search for left and right brackets globally
     //replace all matches with empty strings
     return strValue.replace(objRegExp,'');
}
```

Notice the pattern in bold. Before you run for the aspirin, let me explain. The only thing you really don't know about this pattern is the following:

```
[\(\)]
```

And this is really not hard to understand, once you hear the explanation. You might already know that in JavaScript the forward slash "****" is called a **delimiter**. A delimiter tells JavaScript that a special character is coming up. JavaScript would normally think that special character is doing some kind of action, depending on the context of where it shows up. For example, the "**(**" in regular expressions means something specific, it helps describe patterns. But we **don't** want to use it for that here – we really want to find a bracket! So a delimiter tells JavaScript "ignore what this character normally does here, we want the LITERAL character here." We actually want to find those brackets.

That explains both forward slashes before each bracket. So far the pattern is "match beginning and ending brackets, globally." The only thing left is the square brackets **[** and **]**. These define a "character set." In other words, it says "find any of these characters in the pattern." You see, without those square brackets, the pattern was really looking for "()" and unless the user entered that with NO numbers in the area code, the pattern will never match.

This way, the pattern will match ANY bracket in ANY spot in the string. So even if the user misplaces one, we'll catch it!

We add that to our function:

```
function FormatPhoneNumber(oPhone)
{
     var strNewPhone;
     var strPhone = oPhone.value;
     strPhone = StripDashes(strPhone);
     strPhone = StripSpaces(strPhone);
strPhone = StripBrackets(strPhone);
}
```

Now what we have is a clean string of numbers, and nothing else – perfect.

Putting things back where they belong

Now the rest is fairly easy. We'll use a **switch** control structure to decide what length phone number we have (either 7 or 10). We need to first get the length of the string (the phone number) for this, so we can test against it:

```
var phoneLength = strPhone.length;
```

Then we can use phoneLength in our **switch**:

```
switch(phoneLength)
{
     case 7:
     break;

     case 10:
     break;

     default:
     break;
}
```

So based on how many numbers the user entered, we'll go into one of 3 conditions: 7, 10 or default. If they have made a mistake, and entered the wrong amount of numbers, all they will get back is that string (minus any dashes, spaces or brackets!) So we'll take care of that first:

```
     default:
     strNewPhone = strPhone;
     break;
```

Remember we set up strNewPhone at the start to hold the new formatted string. When the validation runs, it will grab the error. Hopefully though, our user might see the odd looking phone number and fix their error.

Now we get to finally use substring(). This time we're going to put the dashes back in the 7 length phone number:

```
strNewPhone = strPhone.substring(0,3) + "-" + strPhone.substring(3,7);
```

What we're doing here is taking apart the phone number in substrings, and then inserting the dash. Let's take an example number of 555-1212. The first substring matches "555" and then we **concatenate** the string (meaning we add onto) a dash. That's the + `"-"` + part. Then we tack on the rest of the phone number again with substring, which grabs "1212."

Doing the brackets is the same thing, just a bit longer:

```
strNewPhone = "(" + strPhone.substring(0,3) + ") " + strPhone.substring(3,6) +
"-" + strPhone.substring(6,10);
```

In this case we start with a bracket, then add in the zip, end a bracket (and a space, for clarity) and the rest is the same as the above (with different index numbers, of course.)

That only leaves one thing left, and that is to assign the new formatted (or unformatted, for bad numbers) to the text box:

```
oPhone.value = strNewPhone;
```

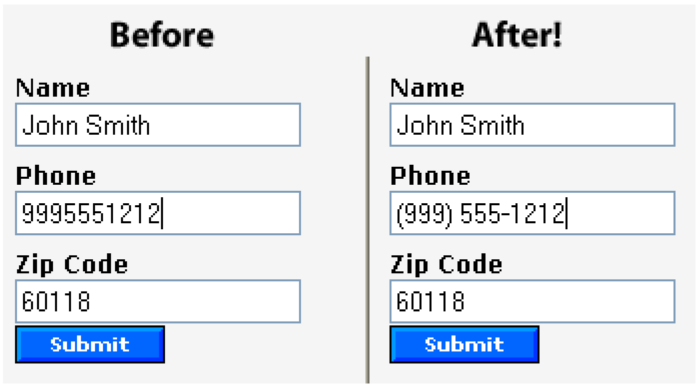
And we're done! The whole function looks like this (including the regex functions):

```
function FormatPhoneNumber(oPhone)
{
     var strNewPhone;
     var strPhone = oPhone.value;
     strPhone = StripDashes(strPhone);
     strPhone = StripBrackets(strPhone);
     strPhone = StripSpaces(strPhone);
     var phoneLength = strPhone.length;

     switch(phoneLength)
     {
           case 7:
        strNewPhone = strPhone.substring(0,3) + "-" + strPhone.substring(3,7);
           break;

           case 10:
        strNewPhone = "(" + strPhone.substring(0,3) + ") " +
strPhone.substring(3,6) + "-" + strPhone.substring(6,10);
           break;

           default:
           strNewPhone = strPhone;
           break;
     }
```

```
    //return strNewPhone;
     oPhone.value = strNewPhone;
}

function StripDashes(strValue)
{
     var objRegExp = /-/g; //search for dashes globally
     //replace all matches with empty strings
     return strValue.replace(objRegExp,'');
}

function StripBrackets(strValue)
{
     var objRegExp = /[\(\)]/g; //search for left and right brackets globally
     //replace all matches with empty strings
     return strValue.replace(objRegExp,'');
}

function StripSpaces(strValue)
{
     var objRegExp = / /g; //search for spaces globally
     //replace all matches with empty strings
     return strValue.replace(objRegExp,'');
}
```

Take a look

And the user would see something like this, if they entered a number like 999 555 1212:



Name	John Smith
Phone	9995551212
Zip Code	60118

Name	John Smith
Phone	(999) 555-1212
Zip Code	60118

Conclusion

You can begin to see the power of what regular expressions can do. They can actually get incredibly complex and can do very powerful operations. Within client side JavaScript they are generally relegated to validation and formatting functions, and are well suited for such.

The regex's I showed you could actually be combined into one pattern, which would be more complex to write. Complex patterns are where regex shines, like validating emails, social security numbers and even credit cards. Of course, those are the more challenging patterns to create correctly.

Using the substring() method comes in handy anyplace where you need part of a string. Build them up and tear them down!

Scripting Radio Buttons without tears.

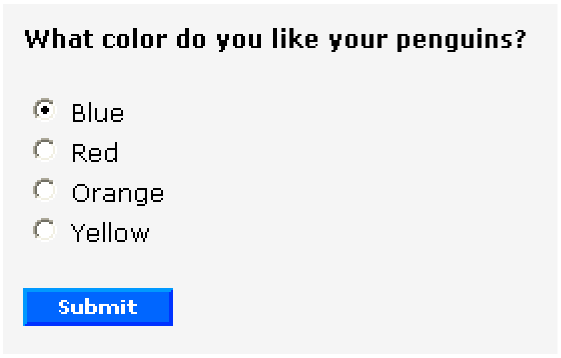
Scripting a checkbox is relatively straightforward. Radio buttons like to throw a wrench into things. Don't let that deter you. In the end if you simply understand what a radio button really is, it will be a breeze to write those scripts you've been itching to write.

Taking a Closer Look at the Radio Button

The better you understand the elements that you are attempting to script, the more success you will have. If I had a nickel for every person who thinks that checkboxes and radio buttons are interchangeable, I'd have at least $5.75. But I digress. The fact is that radio buttons serve a completely different purpose from checkboxes.

Radio buttons perform in **groups**. Let me just say that one more time so the class is clear: radio buttons perform in groups! If you have a form that has one radio button in it, give yourself 39 lashes. Then try and uncheck that radio button once you've checked it. When you realize you cannot, you aren't the first (or the last) to make that mistake.

Radio buttons perform in groups in order to allow the user to choose **one** of the items in the group, but not more than one and not less than one, either. Examine the screen shot below to see the correct use of a simple radio group:



The user can choose one color they like for their penguin. There is a color selected already as well, so one and only one value will be sent with the form.

Let's have a look at the radio button properties.

Table 1. Properties of the Radio Button

Property	**Description**
checked	Indicates whether or not the radio button has been checked (whether by the user or a script), similar to a checkbox.
name	The name of the checkbox which is the "control name" of the element.

type	Indicates the type of input element, in this case a radio button
value	A value you would like the radio button to have, such as "blue" or "purple".

The checked property is pretty self-explanatory. Just remember that not only a user can check or uncheck a radio button, but a script can as well.

The type property is the same we used in our last script to check that the input object was a checkbox before we attempted to check it. In this case, type would return "radio".

The Further Mysteries of the Radio Button

Let's step right up to the plate with the main problem with scripting radio buttons. If you always think of the radio button as part of a group, it will help you remember how to script the object. When you start to think of it as an individual object, you can easily get confused.

In the past, we have been able to get the set value of an object by doing the following:

```
document.forms['myForm'].elements['myFormObject'].value;
```

And that has worked fine. So if we had a text box that had the value of "bob" then the line above would in fact return "bob." Now, if we have the following radio button:

```
<input type="radio" name="penguinColor" value="blue"> Blue
```

And we were to try:

```
document.forms['myForm'].elements['penguinColor'].value;
```

We would NOT get "blue." We would get "undefined." Confused? This is why we don't think of radio buttons as individual items, but as part of a **radio group**. Let's again take a look at the above radio button in the context of the whole form.

```
<form name="question">
<h5>What color do you like your penguins?</h5>

<input type="radio" name="penguinColor" id="bluePenguin" value="blue"
checked="checked" /> <label for="bluePenguin">Blue</label><br />
<input type="radio" name="penguinColor" id="redPenguin" value="red" /> <label
for="redPenguin">Red</label><br />
<input type="radio" name="penguinColor" id="orangePenguin" value="orange" />
<label for="orangePenguin">Orange</label><br />
<input type="radio" name="penguinColor" id="yellowPenguin" value="yellow" />
<label for="yellowPenguin">Yellow</label><br />

<p><input type="submit" name="submit" value="Submit" class="btn" /></p>
```

```
</form>
```

Again we use the LABEL tag for accessibility. In such a small form with only a few elements, the TABINDEX attribute really isn't needed, however remember to use it where appropriate.

The first thing you may have noticed is that ALL the radio buttons have the same name! Is that right? Yes - because they are all part of a **group**. A group shares the same name. It only has a value within the group context. "Blue" is meaningless by itself, until I tell you that it is the color of a penguin.

What's With the Name?

Therefore, because "penguinColor" could be one of 4 values, JavaScript returned undefined. Even had one of the radio buttons been checked, it still would have returned undefined. Simply because the script doesn't know if you are referring to the checked radio button or one of the other three, you didn't specify WHICH penguinColor radio button you wanted.

But how do you do that? Ah, yes, the wonderful array rides to the rescue again! The radio group "penguinColor" has an ARRAY of buttons within it (starting at 0 of course). Remember, we can access a member of an array using brackets (like: []). Often you can use both the array position (0 through however many are in the array) or the name of the item. In this case, each radio button does NOT have a name in the array. However, that won't cause a problem as you will see.

So how can we reference a radio button? Let's assume we want the value of the first radio button in the above group. Here it is:

```
document.forms['myForm'].elements['penguinColor'][0].value;
```

And that will return "blue" for us. Notice, you don't place the dot between the name of the radio group and the array of buttons. Why? Well, that would probably be a very geeky technical discussion that is beyond our scope. Try it, you'll see it works.

The Script

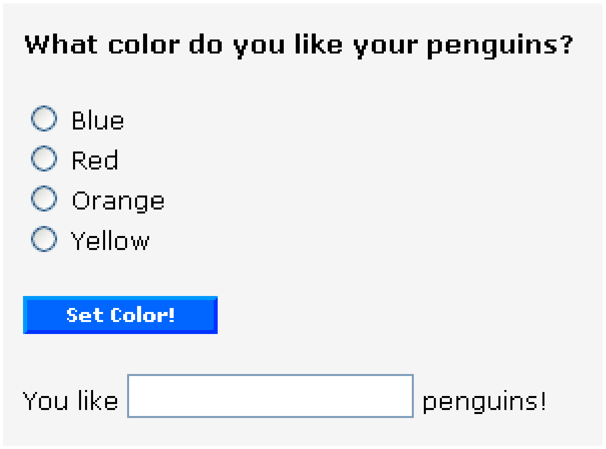
Now we know how to get the value of a radio button, we can use it to do something. In a simple questionnaire, the user would select a single radio button and that value would get sent to the server and so on. Let's pretend we needed that radio button value to do something on the page instead. (You can debate whether that's a correct use of radio buttons with your buddies later).

Let's say we want to take the color the user selected and put it into a text box to display a little message of the selection. It's kind of a "mad lib" type thing and kind of circa 1996, but it will get some basic concepts across. We'll have to take the value of the radio button and put it into our text box. We'll use a button with an onClick event to fire off our script.

Here is our updated form snippet:

```
<form name="penquins">
<h5>What color do you like your penguins?</h5>

<input type="radio" name="penguinColor" id="bluePenguin" value="blue" /> <label
for="bluePenguin">Blue</label><br />
<input type="radio" name="penguinColor" id="redPenguin" value="red" /> <label
for="redPenguin">Red</label><br />
<input type="radio" name="penguinColor" id="orangePenguin" value="orange" />
<label for="orangePenguin">Orange</label><br />
<input type="radio" name="penguinColor" id="yellowPenguin" value="yellow" />
<label for="yellowPenguin">Yellow</label><br />

<p><input type="button" name="showColor" value="Set Color!"
onClick="DisplayColor(this.form);" /></p>

<p>You like <input type="text" name="chosenColor" value="" /> penguins!</p>

</form>
```

Here is what the form looks like now:



What color do you like your penguins?

The first thing you will notice is that I removed the "checked" attribute from the first option, blue. This is because we want the user to actively choose a color, not just accept what is already there. When you validate your form, you can verify that the user has made a choice before allowing them to continue

Secondly, I added a text input called chosenColor to display the color the user selects. Again, kind of a mad lib type thing.

Lastly, I changed the button text for the button that will trigger the script when clicked, and added the function call using the onClick event, which simply means "when I am clicked."

What is THIS?

Remember that the keyword "this" refers to the current object. We are doing the following to trigger our script:

```
onClick="DisplayColor(this.form);
```

Since the onClick event is in the button object, "this" refers to that button. By adding the ".form" I'm simply saying "the form in which this button resides." It's always a good idea to send along the form as a parameter; it makes things easier. It also helps you to abstract scripts, which we will talk about in the future.

Back to the Script

So taking a first look at the script:

```
function DisplayColor(oForm)
{
     var displayBox = oForm.elements['chosenColor'];
     var color = oForm.elements['penguinColor'];
}
```

There are two objects we need to deal with. The first is the new text box in which the color will be displayed. The second is the color the user chooses by selecting a radio button.

Using our form reference we sent as a parameter (oForm, which is an abbreviation I like to use that means "object form" or my form object) I can easily whip up two references to both objects.

The next thing we have to do is get the value of the selected radio button. In this case we aren't concerned about the user not moving the selection off the default. We'll just use the "naked" value if they don't choose a color.

As you might have guessed (since we have to use an array to get to the selected button) we are going to need a loop here, and it is going to be a *for* loop again. To do a for loop we need to know how many times to iterate through the loop. Once again we can use the length method of the radio group, which give us the number of buttons in the array.

```
for(i=0; i<color.length; i++)
{

}
```

Since I set up the variable "color" to point to our radio group, I can say "color.length" and get the number of radio buttons. Since we have five radio buttons, color.length is going to be 5.

Next we have to see if the current button is checked or not. This is going to be done just like we did with the checkboxes. We will simply check the "checked" property to see if it is true or false. If it is true, we know we have the radio button the user selected.

```
for(i=0; i<color.length; i++)
{

     if(color[i].checked == true)
     {

     }
}
```

Hopefully the color[i] reference has not confused you. Remember we set up the variable "color" to point to the radio group with the line:

```
var color = oForm.elements['penguinColor'];
```

So "color" is equal to "oForm.elements['penguinColor']". The [i] on the end is simply the iteration which we are on (or the loop number) in the *for* loop. And of course, the brackets are referring to the radio button array for our radio group. So on the first pass through the loop, color[i] is going to be equal to color[0], which is the first radio button (the default one) in the group.

If that happens to be checked (using the == operator to for equality) then we have our color, so we can set a variable to hold the selected color.

```
for(i=0; i<color.length; i++)
{

     if(color[i].checked == true)
     {
           var penguinColor = color[i].value;
     }
}
```

Notice we introduced a new variable here, right in the middle of the script to hold that color. While that's okay to a point, I like to get my variables defined at the top of the script. Defining everything up top makes your scripts more readable, and that is always a good thing. In this case I didn't want to make things confusing so I saved it until now. A better way would have been to do:

```
var penguinColor;
```

at the top of our script. Then I would have all my variables in one place and things are a little bit neater.

In any case, penguinColor now holds the color that the user selected. The last thing we need to do is set the text box value to that color. This is very simple:

```
displayBox.value = penguinColor;
```

We already defined displayBox as our text box. The variable penguinColor holds the selected color. So setting the value of our text box to penguinColor will put the color into the box. Here is the whole script:

```
function DisplayColor(oForm)
{
     var displayBox = oForm.elements['chosenColor'];
     var color = oForm.elements['penguinColor'];
     var penguinColor;

     for(i=0; i<color.length; i++)
     {
           if(color[i].checked == true)
           {
                 penguinColor = color[i].value;
           }
     }

     displayBox.value = penguinColor;
}
```

Validating the Choice

We have to take care of one last piece of business. What if the user forgets to choose a color before pressing the button? If we did not handle this possibility, the user might get the term "undefined" in the mad lib, and that would be very sloppy indeed.

What needs to be done is to make sure that a choice has been made. We can use a new variable I like to call a **flag**. Its' use is very simple. At the top of the form we set this flag to false:

```
var choiceFlag = false;
```

This tells us that at the beginning of the function, we know that no choice has yet been made. If we get to the end of our function and choiceFlag is still false, we'll know that the user forgot to choose a color and we can alert them.

Now we know when a radio button is checked, simply by looking at this part of the function:

```
if(color[i].checked == true)
{
     penguinColor = color[i].value;
}
```

Since the checked property of the current radio button has evaluated to true, we know a button has been checked. Therefore, we can set our flag to true:

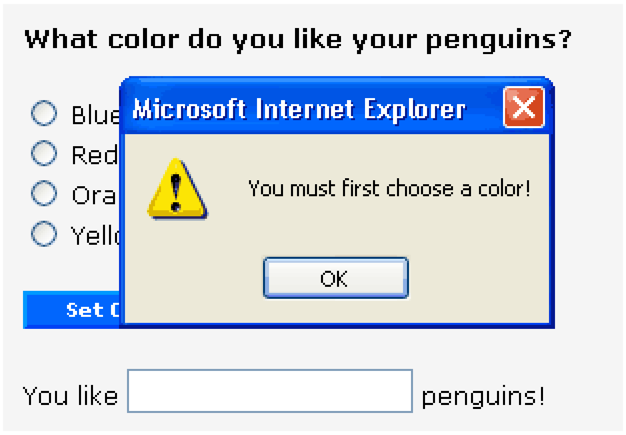
```
if(color[i].checked == true)
{
      penguinColor = color[i].value;
choiceFlag = true;
}
```

So if the script runs through the radio button array and nothing is checked, the flag will still be false. We can test this flag at the end of our script:

```
if(choiceFlag)
{
      displayBox.value = penguinColor;
}
else
{
      alert("You must first choose a color!");
}
```

In our if statement, we simply check that choiceFlag is true. (Note: we could write if(choiceFlag == true), and that would yield the same result. You can write it that way if you understand it better, but both versions mean the same thing). If the flag is true, we know a choice has been made so we set the color. Otherwise, we use the alert method of JavaScript to tell the user they have to choose a color. We don't attempt to set the mad lib color (since we would get an "undefined" in our text box, which we don't want) we simply exit the script while doing nothing. The user then can go in and choose a color and successfully complete the mad lib.

Below is a screen shot of how it would look if the user forgot to make a choice:


What color do you like your penguins?

Microsoft Internet Explorer

You must first choose a color!

You like penguins!

Conclusion

Simply remember to think of radio buttons in groups, and not as single items and you will be well on your way to successful scripting. (If you need an individual on/off control, you want a checkbox. See the paragraph "Scripting Checkboxes"). Remember that referring to a radio button by its control name does not get you its value. The control name (which is name="radioButtonName") refers to the group.

The correct way to get at a particular radio button is to access its position in the radio button group array. Iterating through the radio button group array gives you access to each button in the group, and that is how you manipulate radio buttons.

Scripting Checkboxes

Applications that have some kind of administration or content management functionality generally make generous use of forms to control that functionality. The more complex the application, the more complex the controlling forms. In this paragraph we will look at groups of checkboxes and how to manipulate them within a form.

Your New JavaScript Library

Starting with this paragraph, I will be saving the scripts we build and placing them in a JavaScript library. This will be free for you to keep and use as you wish. Since I will always try to build on real world examples, hopefully you will both learn something about JavaScript and end up having a library of useful scripts at your disposal. Each time another script is going to be added, I'll be sure to make mention of it, and you can download the latest library anytime you wish.

Taking a Closer Look at the Checkbox

The better you understand the elements that you are attempting to script, the more success you will have. I've already talked about the importance of using correct syntax and referencing objects correctly. Those two things alone will save you much trouble.

But what really should come before you attempt to script something is a complete understanding of the object itself. Again, this is where your handy dandy JavaScript or HTML reference comes in. So let's see exactly what the checkbox object is really made of.

Remember that objects have properties, which are current settings (such as a name, an id or a measurement), methods, which are actions (such as print, close, etc.) and event handlers, which specify how an object can react to user or browser interaction (such as when a page loads or when something is clicked.)

Table 1. Properties of the Checkbox

Property	Description
checked	Indicates whether or not the checkbox has been checked (whether by the user or a script)
name	The name of the checkbox which is the "control name" of the element.
type	Indicates the type of input element, in this case a checkbox
value	A value you would like the checkbox to have, such as "yes" or "no".

When an input object, such as a checkbox or a radio button group or even a button is submitted as part of a form, the name/value pairs are sent to the page or script handling the form. This means that if I have a checkbox named "status" and a value for that checkbox set to "off", then the form will send:

```
status=off
```

for that checkbox. The form sends a whole set of these name value pairs for every object in the form that has them. So the name and value properties of the checkbox control those values.

The checked property is pretty self-explanatory, just remember that not only a user can check or uncheck a box, but a script can as well (which is what we will be doing with our script in this paragraph.)

The last property, type, is a very helpful yet overlooked tool. Every input object in a form, whether it's a textbox, a button or a checkbox or whatever has a type associated with it. In this case, it is logically:

```
type=checkbox
```

You can use this to your advantage, as we will shortly see.

The Checkbox Event Handler

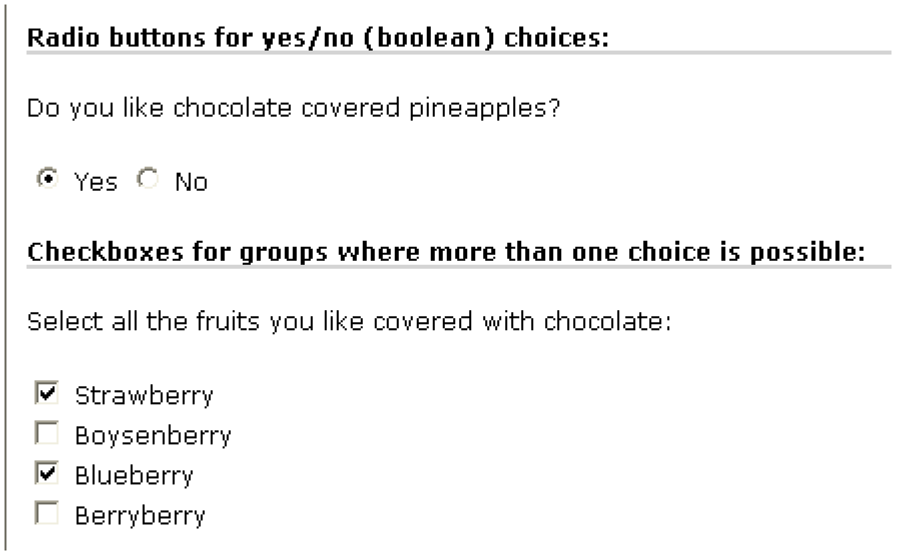
The checkbox has only one event handler, onClick. When you want a script to respond to a user clicking on a certain checkbox, you set it to the onClick event handler like this:

```
onClick = "SomeScript();"
```

A Quick Note About Using Checkboxes

It is important to remember that each form element has a specific purpose. You should be careful about breaking expected conventions with form elements since people expect them to behave a certain way.

A checkbox is generally used to toggle a value between "on" and "off" or "selected" and "unselected" type settings (otherwise known as "boolean" type decisions). When you have a group of say 4 checkboxes, it is perfectly valid for 2 (or all) of them to be checked. With a radio button group, you should only have one button checked.



Radio buttons for yes/no (boolean) choices:

Do you like chocolate covered pineapples?

[✓ Yes] No

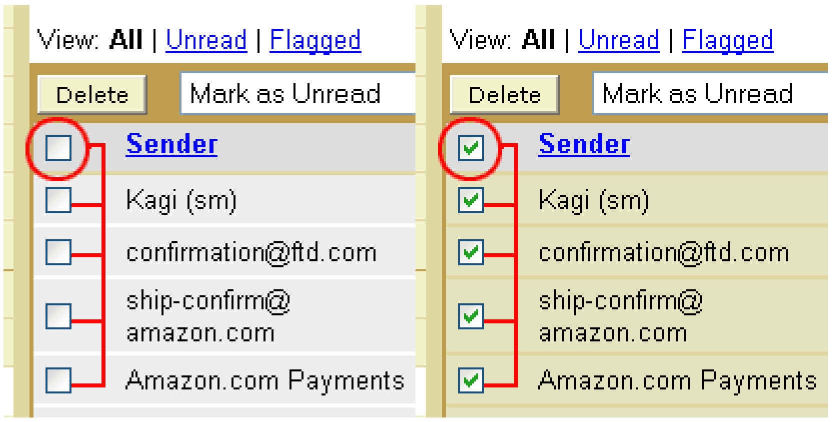
Checkboxes for groups where more than one choice is possible:

Select all the fruits you like covered with chocolate:

- ✓ Strawberry
- ☐ Boysenberry
- ✓ Blueberry
- ☐ Berryberry

Our Script

In our script, we are going to have a group of checkboxes that either select or deselect an item, with a controlling checkbox that toggles them all between a checked or unchecked state. The idea is to save the user some time and clicks by changing the state of all the checkboxes at one time. This will provide a good scripting example as well as being a common real world example. In fact, if you have a Hotmail or Yahoo! email account, you probably already use this very functionality. Take a look at the screen shot below from Yahoo!:



Clicking the "toggle" checkbox next to "Sender" either checks or unchecks all the checkboxes in the list of emails. This is an all-or-nothing operation. One smart feature is that if you have all of the checkboxes selected, and then you deselect one the toggle is unchecked. This is nice because it can save you a click. If you uncheck one (or more) then decide you want them all checked again, you simply click the toggle checkbox and they are all checked again. Without this functionality, you would have had to click the toggle once to uncheck them all, then click it once more. We will build this functionality into our script as well.

We will also include a text box to represent the fact that there are usually other form elements in a typical form than just checkboxes. This is something our script will have to deal with.

The Example HTML

Below is the form we will script:

```
<html>
<head>
(script goes here)
</head>
<body>

<form name="admin">
<p>Which user(s) should have access?</p>
<input type="checkbox" name="allUsers" id="allUsers" value="on"
onclick="ToggleAllUsers(this.form, this.checked);" tabindex="1" />
<label for="allUsers">Select All Users</label>

<input type="checkbox" name="tom" id="tom" value="on"
onclick="Toggle(this.form.elements['allUsers']);" tabindex="2" />
<label for="tom">Tom</label>

<input type="checkbox" name="dick" id="dick" value="on"
onclick="Toggle(this.form.elements['allUsers']);" tabindex="3" />
<label for="dick">Dick</label>

<input type="checkbox" name="harry" id="harry" value="on"
onclick="Toggle(this.form.elements['allUsers']);" tabindex="4" />
<label for="harry">Harry</label>

<br />

Comments: <input type="text" name="comments" size="40" tabindex="5" />
<input type="submit" name="submit" value="Submit" tabindex="6" />
</form>
</body>
</html>
```

As you can see it's a simple form. Assume you are an administrator and you need to grant or revoke access for the three users. The top checkbox will allow the administrator to check all the users at once, by simply checking the "Select All Users" button.

Accessible Forms

You may have seen some unfamiliar HTML in the above code snippet. This code helps make forms more accessible. Any time you are scripting forms you should keep in mind accessibility. Two things that really help are the **LABEL** tag and the **TABINDEX** attribute.

The LABEL tag was introduced with HTML 4 and is supported by IE4+ and NS6+ as well as recent versions of Opera (as far as the major browsers are concerned.) The label tag performs two services:

1. Associates description text with a form control.
2. Increases the size of the active user interface region for selecting the form control. (This relates to Fitts's law, see http://www.asktog.com/basics/firstPrinciples.html#fitts's%20law)

The first service actually provides a link to descriptive text and the form control. The way to do this is very simple. As you may or may not know, the **ID** attribute is a **unique** identifier for an HTML element. You can (and sometimes should) have *both* a NAME and ID attribute for certain HTML elements, especially form elements. As you can see above, each form element has a unique ID attribute that matches its' NAME attribute.

This corresponds to the **FOR** attribute in the LABEL tag. The FOR attribute should match the ID of the form control it is associated with, and this allows the LABEL element to function correctly. You are simply saying, "I want this lable to be FOR my element which has such-and-such ID." So for the toggle button:

```
<input type="checkbox" name="allUsers" id="allUsers"…
```

Note the same NAME and ID attribute. Then the corresponding LABEL:

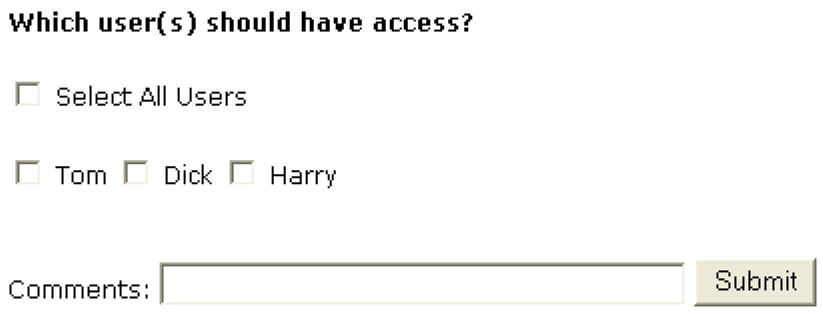
```
<label for="allUsers">Select All Users</label>
```

You can see how the FOR and ID attributes match. This now "enables" the textual descriptions for that element. Now, the user can actually *click on the text* as well as the form element itself and manipulate the control (in this case, check the checkbox). This is extremely helpful for people who have sight disabilities.

The TABINDEX can also be helpful. Notice that each element in the form has a TABINDEX attribute with a number. The number corresponds to the order in which the form elements get "focus" (i.e., the browser gives control) when the tab key is pressed. With simple forms, as in our case, this didn't actually change the default behavior. However, in more complex forms this can be helpful.

IE4+ will also allow you to attach an **ACCESSKEY** attribute to your LABEL element. This is in essence a "shortcut" key to the element itself. While not fully supported, when well used this can be a nice accessible feature in a form.

Here is how the form looks:



Scripting the Toggle Function

You probably noticed the onclick event handler with our function already there, named ToggleAllUsers(this.form). Notice we are passing the form along as a parameter, so we have that reference handy. Remember the "this" keyword simply means the current object. The checkbox is the current object (this) and we want its form (the form it resides in). When that checkbox is clicked, our function will get executed.

As you might have guessed, the key to checking all the checkboxes is to loop through our form, and check each one. The form object has a length method which will give us its total number of elements. We can then use that number to loop through the form. Let's set that up first:

```
function ToggleAllUsers(myForm)
{
     var formLength = myForm.length;
     for(i=0; i<formLength; i++)
          {

          }
}
```

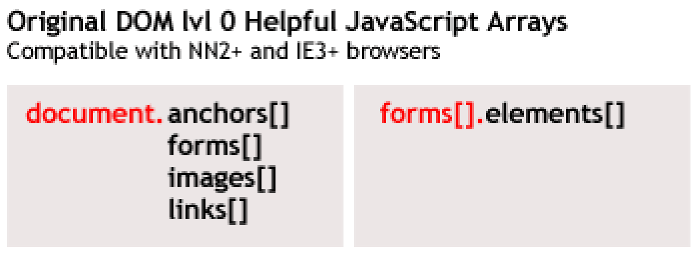
Remember a for loop has three conditions: the initial value (usually set as 0), how long to continue looping (in our case, as long as i is less than the length of the form, which is 6, we keep going), and by what value to increment the loop (usually by 1 - remember: i++ is shorthand for i= i + 1).

Now we can do something inside our for loop 6 times, once for each element in the form. The problem is, every element is not a checkbox. We only have four checkboxes, the other two elements are a text box and the submit button. We can't very well check those, since those elements don't have a checked property. (Even if they did, we wouldn't want them checked!)

The Type Property

Here is where the type property comes in mighty handy. As noted in table 1, the type property of a checkbox is, obviously, checkbox. We can first check each element to see if it is of the correct type. If it is, then we can check it.

Remember, the best way to access form elements is the elements array, which is itself a part of the forms array. It's easy to get all these arrays confused, so let's look at a small chart to make it clearer:



Original DOM lvl 0 Helpful JavaScript Arrays
Compatible with NN2+ and IE3+ browsers

document.anchors[] forms[] images[] links[]	forms[].elements[]

Reference an element by array position or NAME attribute:

document.images[0] or document.images['daffyDuck.gif']
document.forms[0] or document.forms['myForm']

document.forms[0].elements[0] **(not recommended)** or
document.forms['myForm'].elements['myElement'] **(recommended)**

So a correct reference to the Tom checkbox is:

```
document.forms['admin'].elements['tom'];
```

Again, an element can be referenced by its name or its position in the array. In this case we want to use its position in the array since we don't want to point to a particular element, we want to point to them all, by looping through them one at a time, to check and see if they are checkboxes. Since our for loop is using the variable i, we will use that variable as the array position:

```
document.forms['admin'].elements[i];
```

Now, each iteration through the loop, we get the first element, the second, all the way to the last. We can shorten this more though, because we already have a reference to our form (remember, we passed it as a parameter). So we can do it this way:

```
myForm.elements[i];
```

To access the type property:

```
myForm.elements[i].type;
```

Now that we have that cleared up, we'll test each one of those elements to see if it is a checkbox Remember to use two "==" signs, which is the test for equality, not one equals sign, which is how you set a value). If it is, then we will set the checked property (which says whether or not a checkbox is checked – true or false) to true, thus checking it. This will check all of our checkboxes!

```
function ToggleAllUsers(myForm)
{
     var formLength = myForm.length;
     for(i=0; i<formLength; i++)
         {
           if(myForm.elements[i].type == "checkbox")
           {
                 myForm.elements[i].checked = true;
           }
         }
}
```

Now, try it for yourself, click the top checkbox and viola! All your checkboxes are checked.

Please, No One Trick Ponies

Well, that's a good start but what if I want to uncheck them all now? I have to do them all manually again. Not so big a deal if I only have 3 users, but what if I have 20 or 30?

What would be really nice is if the top checkbox were actually a toggle, not just an on switch. Just like a light switch in your house turns the light on and off, not just on.

Let's change our top checkbox text to say "Toggle All Users" instead of "Select All Users."

Now, how do we know whether all the boxes are checked or not? It's surprisingly easy. We know that if the "Toggle All Users" checkbox is checked, then in fact all the users are checked, right? Since our function checks all the boxes when the toggle checkbox is checked. Therefore, we can simply test that checkbox's checked property. If it is true (it will return either true or false, just like you can set it to true or false) we know we need to uncheck all the boxes. If it is false, we need to check them.

Note: You are now undoubtedly asking "what if the user checked them all then unchecked one (or more)? Then the top box would be checked but they would NOT all be checked. As mentioned earlier, this would cause us to have to make an extra click to recheck them all.

All we need to address this is a very simple function. We'll take care of that right after we finish the main function.

Passing the parameter is easy. We are already using the **this** keyword to pass the current toggle checkbox's form. We can again use the **this** keyword (which says, 'I am the current object') and attach the checked property to it.

```
onclick="ToggleAllUsers(this.form, this.checked);
```

Now our second parameter sends the state of the toggle checkbox, true or false. Let's name that parameter something that makes sense, like **isChecked**. Then, we'll test that parameter's value. If it is true, as we said, we'll uncheck all the boxes, and vice versa. When using true false values, we can simply test for true like so:

```
if(myValue)
```

If myValue is true, the expression above will evaluate to true, and continue on into the if brackets. If it is false, it will of course evaluate to false and skip over the if statement altogether. So if its true we'll do what we already did, set them all to true. If not, we'll add an else clause and uncheck the boxes instead.

```
function ToggleAllUsers(myForm, isChecked)
{
	var formLength = myForm.length;
	for(i=0; i<formLength; i++)
	{
		if(myForm.elements[i].type == "checkbox")
		{
			if(isChecked)
			{
				myForm.elements[i].checked = true;
			}
			else
			{
				myForm.elements[i].checked = false;
			}
		}
	}
}
```

Notice we changed our function name to reflect the new functionality. Try the function out. You know have your very own checkbox switch!

Unchecking the Toggle

Now, if the user selects all the checkboxes, then unchecks one or more, we want to *uncheck* our toggle to reflect the correct state of affairs. This is quite simple, remember all we have to do is set the checked property to false to uncheck a checkbox.

So on each of our checkboxes (other than the toggle, of course) we'll add a simple function called Toggle() that merely unchecks our toggle checkbox.

```
function Toggle(toggleBox)
{
     toggleBox.checked = false;
}
```

Notice, we are going to pass the toggle checkbox as a parameter to the function. Why? Two reasons:

1. This way we don't have to make any long references to the element itself, we only need to attach the property (checked) to the element reference (in this case, toggleBox)
2. We maximize code re-use. If we were to use an actual form element name here, then every time you wanted to use this function somewhere else you would have to go in and change the name. This way you don't. You can grab this function any time and use it in any application simply by passing it an element name.

So add this to each checkbox like so:

```
onclick="Toggle(this.form.elements['allUsers']);"
```

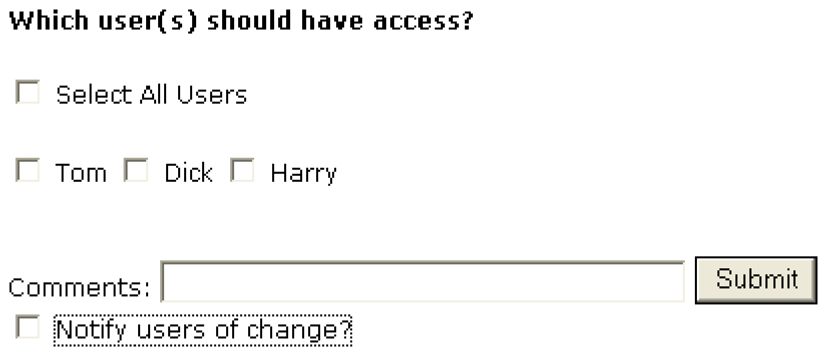
In passing the element reference we make the same use of the elements array we would in a function. We use the ever present **this** keyword with the form attached (meaning "I want to use this form, the one this element resides in"), and using the elements array we specify the name of the element.

One More Wrench

Ah, but you say "what if I also have yet another checkbox in my form, and it isn't one of the user checkboxes?" Excellent question! Let's modify our form by adding the following checkbox after the text box:

```
<input type="checkbox" name="notify" id="notify" value="yes" /> <label
for="notify">Notify users of change?</label>
```

This makes our form look like:



Let's say this checkbox tells the system whether or not it should send an email to all the affected users telling them their access level has been changed. This checkbox value is completely separate from the user access checkboxes.

The problem is, our toggle function will toggle this checkbox too, and we don't want that. On the one hand, this is a pretty simple problem to solve. Having a simple form makes it easier. We'll look at an easy solution and then offer a suggestion on another one that might handle a more difficult form.

Since we know the name of the checkbox we don't want to affect, we can simply test for it when doing the checking and unchecking of the boxes:

```
if((myForm.elements[i].type == "checkbox") && (myForm.elements[i].name !=
"notify"))
```

We've added a logical and operator, which says "first, see if I have a checkbox" AND then "if this checkbox name is NOT (!=) 'notify', go ahead and check or uncheck boxes". With a logical and, BOTH conditions must be met for the script to proceed.

Try it yourself. You could add two or more and conditions if you had two or more boxes you didn't want to check.

However, if you had, say 10 or 20 checkboxes you didn't want to check, you would not want to have 10 or 20 and conditions. At that point you would probably want to build yourself an array of checkbox names you didn't want to check, and then step through that array making sure you were not checking them. However in this simple example, this method works fine.

Here are the first two functions for our JavaScript library:

```
/*
ToggleAllUsers:
     A function that checks/unchecks all checkboxes when a toggle is clicked
Usage:
     Create an additional checkbox in your form. This will be the "toggle"
     (on/off) switch. This checkbox should call the function using the onclick
     method. Function will loop through all checkboxes and check/uncheck
     them based on whether or not the toggle is checked.
Parameters:
     Send the current form and the status of the toggle checkbox,
     whether checked or not (true or false). Example:
     onclick="ToggleAllUsers(this.form, this.checked);"
Notes:
     Additionally, if you have a checkbox in the group that should NOT
     be checked, you could exclude it in the first IF loop. If the name
     matches the checkbox you wish excluded, do not enter the loop. Example:
     if((myForm.elements[i].type == "checkbox") && (myForm.elements[i].name !=
"notify"))
Gotchas:
     Note that the isChecked property is checked AFTER the action of clicking
it.
     So if you had clicked the toggle when it was not checked, it will send
TRUE
     as the value for isChecked.
*/
function ToggleAllUsers(myForm, isChecked)
{
     var formLength = myForm.length;
     // loop through each element in the form
     for(i=0; i<formLength; i++)
     {
           // if the element is a checkbox, toggle it
           if(myForm.elements[i].type == "checkbox")
           {
                 // The toggle is now checked (it was clicked when unchecked)
                 // so check them all
                 if(isChecked)
                 {
                       myForm.elements[i].checked = true;
                 }
                 // the toggle is unchecked, uncheck them all
                 else
                 {
                       myForm.elements[i].checked = false;
                 }
           }
     }
}

/*
```

```
Toggle:
     Companion function to ToggleAllUsers. Unchecks the toggle.
Usage:
     When one of the group checkboxes is checked, uncheck the toggle.
     This will save a click if the user wants to recheck them all.
Parameters:
     Send the toggle box as a reference to keep the function abstract.
*/
function Toggle(toggleBox)
{
     toggleBox.checked = false;
}
```

Taking it further with DHTML

Modifying page elements on the fly

In the "old days" web developers did everything – and I mean everything – in the local page, right in the code. That meant presentational tags all the way down to JavaScript functions spread throughout the page. We soon found out that was the wrong way to do things. Maintenance is a nightmare and consistency throughout becomes non-existent. Even worse, errors easily worked their way into such a development strategy.

In recent times, we've seen the introduction of XHTML and CSS to control the presentation of web pages. It now seems that almost everyone recognizes the benefits of separating presentation code from the content itself. Now developers are realizing that the same benefits can be had with the "behaviour" layer – the JavaScript layer.

In other words, we can do more than just link our JavaScript files instead of putting them locally in the page. What we can do is begin to remove all the function calls and bits and pieces of JavaScript from our pages and instead move the control to the functions themselves. This separates our code layers even further and makes our JavaScript easier to maintain and more powerful.

getElementsByTagName

If we are going to remove JavaScript from our code in our local pages, then we need a powerful way to get at things in large chunks. Fortunately, the DOM Core Level 1 provides just such a tool called **getElementsByTagName**. The W3C explains it like so:

Returns a NodeList of all the Elements with a given tag name in the order in which they would be encountered in a preorder traversal of the Document tree.

Parameters
    `tagname` - The name of the tag to match on. The special value "*" matches all tags
Return Value
    A new NodeList object containing all the matched Elements.

That sounds a bit technical in typical W3C style. The first thing talked about is a NodeList. A NodeList is an ordered collection of nodes – which is similar to an array. Remember that an array looks like this:

```
vehicleArray

[0] Car
[1] Truck
[2] Van
[3] Sled
```

And you could use various methods to manipulate the data in such an array. An ordered collection is almost the same thing. It's simply a collection of the nodes that match the element you specified as the parameter. And there are various ways you can then get at that information. This is exactly what we need.

Example 1

That might not exactly come across as easily understandable without an example. Let's say we want to get at all the DIV elements on our page. It's easier than you think:

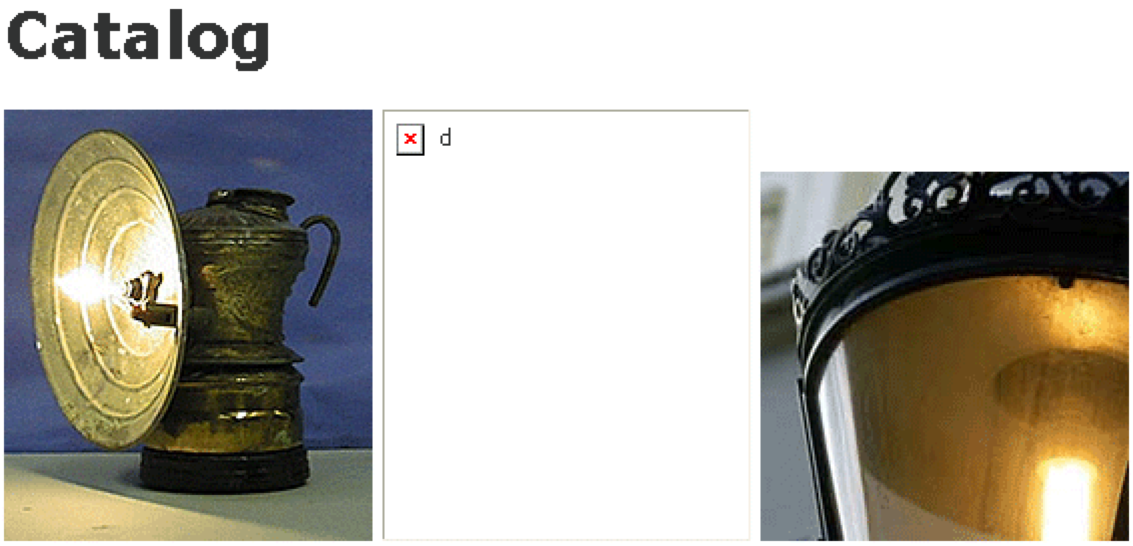
```
var myDivs = document.getElementsByTagName("div");
```

When the above line is executed, the variable `myDivs` will now hold that ordered collection we talked about. Essentially a NodeList (simply a list of nodes – or in this case DIVs) stacked one after the other. If we have 10 DIV tags in our page, then `myDivs` now holds an ordered collection of 10 DIVs. This collection has one method and one attribute we can use. The method is `item` and the attribute is `length` – which we will use. The length of a NodeList or ordered collection is just like the length of an array.

That might help a little but let's move on to a real world example. Christopher Bruno asked a question that we can use as an example on DMXzone.com:

> "I'm new to this stuff & am having a tough time understanding how to write a basic if/then condition in an ASP/Javascript page in DreamweaverMX. My page contains a list of records from a db. I have a thumb image associated w/each row. whose source value is set by referencing the image location & name, as determined by a value from the dataset, as follows:
>
> <img src="images/properties/<%=(rsListings.Fields.Item("mlsid").Value)%>_thumb.jpg" alt="noPhoto" name="listingPhoto" width="36" height="39" border="0">
>
> This works fine, but I need to set the src value differently if there is no image for the record, as determined by the value of the "webImage" field in the data set."

His main problem is that he has an image coming out of a data source (let's assume it is a product catalog for lamps) where he actually doesn't have the picture yet. The product exists and it is in the database – but for some reason the picture is not ready. However, the server side script is writing out an image tag for the product as it should, and the user is seeing a broken image. That's a bit unprofessional, it would be better to show them a "no image available" image instead. So right now we have this:



As you see the middle lamp has no image.

I'm not sure why you would write out "noPhoto" as an ALT value. Instead, let's assume that the server side script does not write out a SRC value or a ALT value for the IMG tag. If there is a photo, then the ALT value is written out as a short description of the picture ('old lamp' for example.) This gives us all we need to work with.

The Catalog Page

Let’s begin by taking a look at what would be the rendered HTML for the catalog page above:

```
<body>

<h1>Catalog</h1>
<img src=“carbidelamp.jpg” width=“184” height=“215” alt=“carbide lamp”
border=“0” />
<img src=““ width=“184” height=“215” alt=““ border=“0” />
<img src=“old-lamp.jpg” width=“184” height=“184” alt=“old lamp” border=“0” />

</body>
```

Notice that the middle image has no SRC value since there is no image available. There is also no ALT value. Somehow we need to fix this with our script. This is where getElementsByTagName will save the day. And the parameter we will use is – you guessed it – the IMG tag.

Beginning the Script

```
function FixImages()
{
     var pageImages = document.getElementsByTagName(‘img’);
}
```

The first thing we need is a reference to every image in the page! Hey no big deal right? All we do is set up a variable (pageImages) and use the getElementsByTagName method. Notice that we attach it to the document – since we want the images in the document. So now, pageImages holds a reference to all our images – in this case – 3 images.

That’s all fine and dandy, but we need to be able to do something with those images. So the first thing we want to be able to do is step through each image to see what it has going on and have a way to manipulate it. We can do this with a FOR loop, just like we would with an array or any other group of items.

```
for(var x=0; x<pageImages.length; x++)
{

}
```

Remember earlier we said that the NodeList has an attribute of length, and this will of course give us the length (how many items) of our pageImages collection (a number we know to be 3). So this FOR loop will run three times and hit each image.

We also said that if a product had no image uploaded, the SRC and ALT values would be empty. So in order to find out if a particular image is broken (i.e., not there) we can check the value of one of those attributes. If it is empty, we know that the image is busted.

We aren't going to want to check the SRC value, because JavaScript won't see that as empty even if there is no value. It sees a relative path up to the empty slot, but it still seems to see something. But we can check for the ALT value, which is not tied to anything at all. So let's set up a variable in the loop to hold the value of the current item's ALT value:

```
for(x=0;x<pageImages.length;x++)
{
     var imgAlt = pageImages[x].alt;
}
```

Since we are in the FOR loop, we use x to indicate what position in the collection we want to look at, and simply attach the alt reference. Now imgAlt holds the current image's ALT value. First though, let's add a variable that holds the image path to our "no image available" image:

```
function FixImages()
{

var noImgPath = "no-image.gif";
var pageImages = document.getElementsByTagName('img');

     for(x=0;x<pageImages.length;x++)
     {
           var imgAlt = pageImages[x].alt;
     }

}
```

Since I am using all my images in the same folder, there really is no path. You of course would put the actual path on the server to the image you would use. Now, let's test to see if the ALT value is empty:

```
function FixImages()
{

var noImgPath = "no-image.gif";
var pageImages = document.getElementsByTagName('img');

     for(x=0;x<pageImages.length;x++)
     {
           var imgAlt = pageImages[x].alt;
           if(imgAlt == '')
           {
                 pageImages[x].src = noImgPath;
           }
     }

}
```

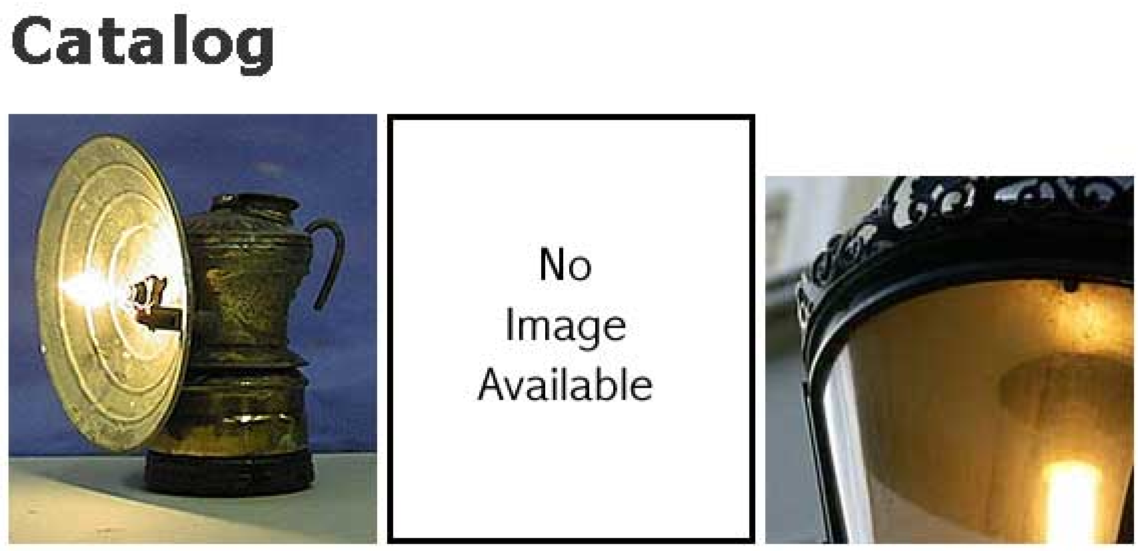
Notice this is all happening INSIDE the FOR loop – since we need to check EACH image. We set up a simple IF statement that checks to see if the imgAlt variable is empty. If it is, the script will enter the clause and we set the current image SRC (pageImages[x].src) to our noImgPath variable – which points right to our “no image available” image! What is the result?

Well, first we have to run the script. In order to do that let’s attach it to the onLoad event. We can’t start running a function on the images if the page is not fully loaded yet – some images may not be loaded. Then our script will generate errors and we don’t want that!

```
window.onload = FixImages;
```

Notice that we did NOT put this in the HTML code on the actual body tag. Remember we are trying to avoid putting JavaScript code in the local page! Also notice we do not use the brackets when referencing a function in this manner. If you included the brackets here, the function would actually try to execute on the spot, and again, would generate errors. We just want to tell the onLoad handler which function to use when it’s time. So the function name without the brackets is a reference TO the function, not the function call itself.

So, what is the result when we load our page?



Ah! Much better. Certainly we want to actually get an image for our product, but this is much better than displaying a broken image.

Now take a good look at the complete page with all the code:

```
<!DOCTYPE html PUBLIC “-//W3C//DTD XHTML 1.0 Transitional//EN”
        “http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd”>

<html>
<head>
     <title>Beginning JavaScript</title>
     <script type=“text/javascript” src=“FixImages.js”></script>
```

```
</head>

<body>

<h1>Catalog</h1>
<img src=“carbidelamp.jpg” width=“184” height=“215” alt=“carbide lamp”
border=“0” />
<img src=““ width=“184” height=“215” alt=““ border=“0” />
<img src=“old-lamp.jpg” width=“184” height=“184” alt=“old lamp” border=“0” />


</body>
</html>
```

Notice that there is not one spot of JavaScript in any of the HTML code – other than our script tag pointing to our script of course. We didn’t have to put any event handlers on the images, on the body tag or anything else like that. So when it comes time to maintain or adjust this script, you simply open the JavaScript file, fix it, and you are done. The fix works on every page that calls the function. Nice and neat.

Conclusion

Not only is this a clean way that helps you maintain things easier, it’s also a smarter way to program. It took less time and effort to write a script like this then doing things the old way. And it is much more powerful as well. You’ll find that we can do lot’s of things this way – adding events dynamically to links on a page, adding content dynamically, and more. With a little creative thinking, you should be able to come up with a few ways where this can help you clean up your code.

Working with the Event Listener

When the DOM came into prominence and DHTML became a viable web technology we seemed to gain a lot of new tools. The great thing about a lot of the new tools is that they are very useful. The downside is that some of them force you to delve a little deeper into JavaScript and maybe learn something new. In other words, you have to step outside your comfort zone. Trust me, this is a good thing!

The event listener was added when the W3C event model was introduced. But first let's make sure we know what an event is – what indeed is an event? Someone once said "without events there are no scripts" and that is pretty much correct. Take a look at your scripts and HTML pages and you'll notice that in order to run your functions, you are triggering them via events.

What are events?

An event allows the user to interact with the web page, and allow JavaScript to react to that interaction. Events are what allow JavaScript to truly make pages interactive. JavaScript needs a way to detect what the user has done in your pages, and then respond to them accordingly. When a user interacts with a page, an event is generated. Sometimes, the page itself generates an event, not the user. For example, the load and unload events are triggered by the page, not the user (even though the user probably caused the page to load/unload, it is technically triggered by the page).

What about event handlers?

So how does JavaScript allow us to handle these events? Since Netscape 2 we've been able to handle events with an event handler. You've certainly been through the source of an HTML page and seen many events attached to HTML elements, such as onMouseover, onFocus and onChange. And even though we are moving toward an "unobtrusive JavaScript" model where these handlers are not going to be directly in the HTML – the concept is still the same and they will still work the same way. These event handlers tell JavaScript what to do when the event is fired.

We will still attach event handlers to elements, we'll just do it in a more efficient manner. We're now able to attach event handlers directly to elements using new methods that will keep our HTML free of clutter – and make our scripts more powerful. For example, suppose you wanted to use TWO events on one element. With the new methods you could add one, listen for the event and handle it, then remove the first one and add a second one. That would be extremely difficult to do the old way.

The only issue we'll have to deal with is different event models, because IE and Mozilla handle things differently. Not to worry though, we'll use object detection and it will be as simple as pie.

Registering event handlers

Now that we have a bit of background, I can tell you that there are 3 ways to register (or attach) an event handler to an element. The first way does not involve the event listener, but we'll take a look at it anyway. The other two ways are the event listener models of IE and Netscape respectively, which is the main crux of this paragraph.

Attaching event handlers to elements using event property binding

A simplified manner of attaching an event handler is to simply add it as a property of an element (called event property binding since you are binding the event to the object property). For example:

```
function Foo()
{
     var myElement = document.getElementById(“foo”);
     foo.onclick = FooFunction;
}

function FooFunction()
{
     //do some stuff when the user clicks
}
```

This is in fact a cross-browser compatible method to attach an event handler. You might wonder why we aren’t using this method instead of the event listeners which are not cross browser compatible. The reason is because the code is not quite as elegant or powerful, and not quite as versatile. As you work more and more with events you’ll see this for yourself. This is still a better way than the old event handlers in the HTML code way, however.

Also notice as we have said in the past - there are no brackets supplied for FooFunction – this is VERY important to note. We want to tell the event handler which function to execute when the event fires. If we include the brackets – the function (FooFunction) will immediately be executed and the **result** of that function will be attached to the element. That is not what you want! Think of leaving off the brackets as “pointing” to the function itself without allowing the function to execute.

Our friend, the anonymous function

We’ve talked about this before but it bears repeating because it’s another good tool to have in the toolbox. If we have a very small function (like say our FooFunction, which maybe has one line of code) then we have a good place to use an anonymous function. It’s called that because it has no name (makes sense right?) Here’s an example:

```
function Foo()
{
     var myElement = document.getElementById(“foo”);
     foo.onclick = function() { this.backgroundColor = ‘#ffc’; }
}
```

This is a little cleaner and easier to deal with. We can use the this keyword to refer back to our current object (foo). Keep this in mind for small functions. The anonymous function also lets us run TWO functions on one element should we desire to do so:

```
function Foo()
{
     var myElement = document.getElementById(“foo”);
     foo.onclick = function() { functionOne(); functionTwo(); }
}
```

You don't often have to do that, but if you do this comes in mighty handy.

Using event listeners

The most powerful and modern way to attach event handlers is the event listener. We have two models to use, the IE and Netscape/Mozilla/Gecko/W3C versions. We'll take the Netscape version first. The method is called addEventListener() and it has three arguments:

Netscape/W3C Model

```
element.addEventListener("eventType", listenerFunc, useCapture);
element.removeEventListener("eventType", listenerFunc, useCapture);
```

As you can see we can both add and remove event listeners. The methods are aptly named because they do exactly what they say: they "listen" for events.

Explaining the arguments

The first argument you supply to the method is "eventType". In other words, which event are we listening for – minus the "on" qualifier. So if we are listening for onMouseout – then the eventType is "mouseout" in ALL LOWERCASE – very important.

Secondly we have "listenerFunc" which simply stands for listener function – what function are you pointing to for use with this event handler? The same rules apply here about not using brackets. The last argument, "useCapture" has to do with event bubbling which is beyond our scope of discussion. It is a Boolean true/false value. For our purposes we'll keep it at false for now. False is going to be your natural default anyway for many or most applications.

So if we were redoing our code above to use the event listener, it would look like this:

```
function Foo()
{
     var myElement = document.getElementById("foo");
     foo.addEventListener("click",FooFunction,false);
}

function FooFunction()
{
     //do some stuff when the user clicks
}
```

Now foo has the event handler onClick attached, and will respond with the FooFunction() function when a user clicks the element. As you might have guessed, we can also use the anonymous function here too:

```
function Foo()
{
     var myElement = document.getElementById("foo");
     foo.addEventListener(“click”, function() { this.backgroundColor = ‘#ffc’;
} ,false);
}
```

IE Model

The IE Model is very similar. The terminology is different, Microsoft uses the terms “attach” and “detach”. The format is like so:

```
element.attachEvent(“eventName”, functionRef);
element.removeEvent(“eventName”, functionRef);
```

Explaining the arguments

Microsoft uses “eventName” instead of type. And IE wants to see the whole event name INCLUDING the “on” part, so you need to keep those two straight. The second argument should be fairly obvious – “functionRef” is the function that will handle our event. It actually makes sense to refer to it as a “ref” or reference because as we explained, that is what we are doing – referring to a function, not executing it on the spot. To continue with our example for IE, it would look like so:

```
function Foo()
{
     var myElement = document.getElementById(“foo”);
     foo.attachEvent(“onclick”,FooFunction);
}

function FooFunction()
{
     //do some stuff when the user clicks
}
```

There are two issues with the Microsoft model. The first one has to do with event bubbling and the set capture which is again out of scope. However the second one is a bit annoying, and makes the this keyword useless. The problem is the event handling function is referenced, not copied, so the this keyword always refers to the window. So in your scripts using the IE version, you’ll either have to work around the issue if you need to use the this keyword, or go back to the event property binding way of doing things. While this can be an annoyance, in general you’ll find you can get around the issue and people are using event listeners more and more.

Putting the two together

So what we ought to do is build a function that determines which version of event listener we need and acts appropriately. Using object detection we can use something like this:

```
function addEvent(myObject, myEventType, myFunction)
{
     // for netscape type browsers
     if(myObject.addEventListener)
{
```

```
            myObject.addEventListener(myEventType, myFunction, false);
            return true;
     }
     // for IE
else if (myObject.attachEvent)
{
            var r = myObject.attachEvent(‘on’ + myEventType, myFunction);
            return r;
     }
     // return false for anything else unsupported
else
{
            return false;
     }
}
```

As you can see we are using object detection – not faulty browser detection – to decide which code branch to run. The addEvent() function has three arguments: the object we want to attach a event handler to, the type of event (such as onmouseover or onclick) and the name of the function that will handle the event.

So, only Netscape type browsers will enter the

```
if(myObject.addEventListener)
```

code branch, and only IE browsers will enter the

```
else if (myObject.attachEvent)
```

code branch. If neither of these objects are supported (maybe some kind of mobile device or unsupported browser hit the page) then nothing is attached, we just return false in the last else clause.

The statements inside each clause should make sense to you since we just went over them. We call the appropriate method and attach/add the event handler. The only thing new here are the return statements.

The return statements are there simply to let the calling function know that something was or was not added to the element. The calling function may or may not make use of the return value, but it’s there in case it is needed. Obviously if the user agent does not support either method and the third clause is entered, false is returned automatically.

How would this be used?

So in the ‘real world’ how would this addEvent() function be used? Let’s take a code snippet from an actual project I am currently working on:

```
this.addElements = function addElements()
     {
           // find all elements that have a span attached
           var pageSpans = document.getElementsByTagName(“span”);
           for(var i=0;i<pageSpans.length;i++)
           {
                 // but grab only the ‘help’ spans
                 if(pageSpans[i].className == “help”)
                 {
                       // attach events
                       addEvent(pageSpans[i], ‘mouseover’, show);
                       addEvent(pageSpans[i], ‘mouseout’, hide);
                       addEvent(pageSpans[i], ‘focus’, show);
                       addEvent(pageSpans[i], ‘blur’, hide);
                 }
           }
     }
```

I have what we are calling a dynamic help tip system in an application we are writing. We wanted a way to test out a help tip system that would be triggered using mouseover and mouseout. The code above is part of a class I wrote that dynamically builds both the link to turn on the system and the help tip box itself, among other things. It works by scouring the page for SPAN elements that have the class “help” attached to them. If the script finds such a SPAN element, guess what it does? It uses addEvent() to attach the event handlers to the area surrounded by the SPAN so it becomes available as a help tip.

The first line is adding the function addElements to my class. It builds a collection (just as we have been talking about using getElementsByTagName()) of SPAN tags. The IF loop checks for SPANs with the class “help” attached, and if it is, then addEvent() is run – attaching the mouseover and mouseout events and tying them to the hide and show functions (as well as blur and focus events).

We can whip together a little example to show you exactly what is going on. We’ll supply some things that the help class is building for simplicity:

```
<h1>Page Title</h1>
<span class=“help”><p>Paragraph One</p></span>
<p>Paragraph Two</p>

<div id=“helpdiv” style=“border: 1px solid black; background: cyan; padding:
10px; display: none;”>This is a help box</div>
```

Notice there is no JavaScript in the HTML. (Normally we would use a stylesheet too, not local CSS!) Add two simple functions to hide and show our help container:

```
function hide()
{
     var oContainer = document.getElementById(“helpdiv”);
     oContainer.style.display = “none”;
}
function show()
{
     var oContainer = document.getElementById(“helpdiv”);
     oContainer.style.display = “block”;
}
```

And we can even use addEvent() now to fire up our span-seeking function that adds event handlers when the page loads:

```
addEvent(window, “load”, addElements);
```

Now the whole thing is a working example. Mouse over paragraph and two and see what happens. Viola! All without any JavaScript on the P tag – it all happens in your script. It may not be pretty but you should be able to recognize the power of these excellent methods.

Conclusion

Using event listeners is another powerful way to make good use of the DOM, and also practice unobtrusive JavaScript. Start making it a habit to build your functionality in your behavior layer – in your scripts themselves, and not in the presentation layer – and you will find the benefits enhance your work greatly.

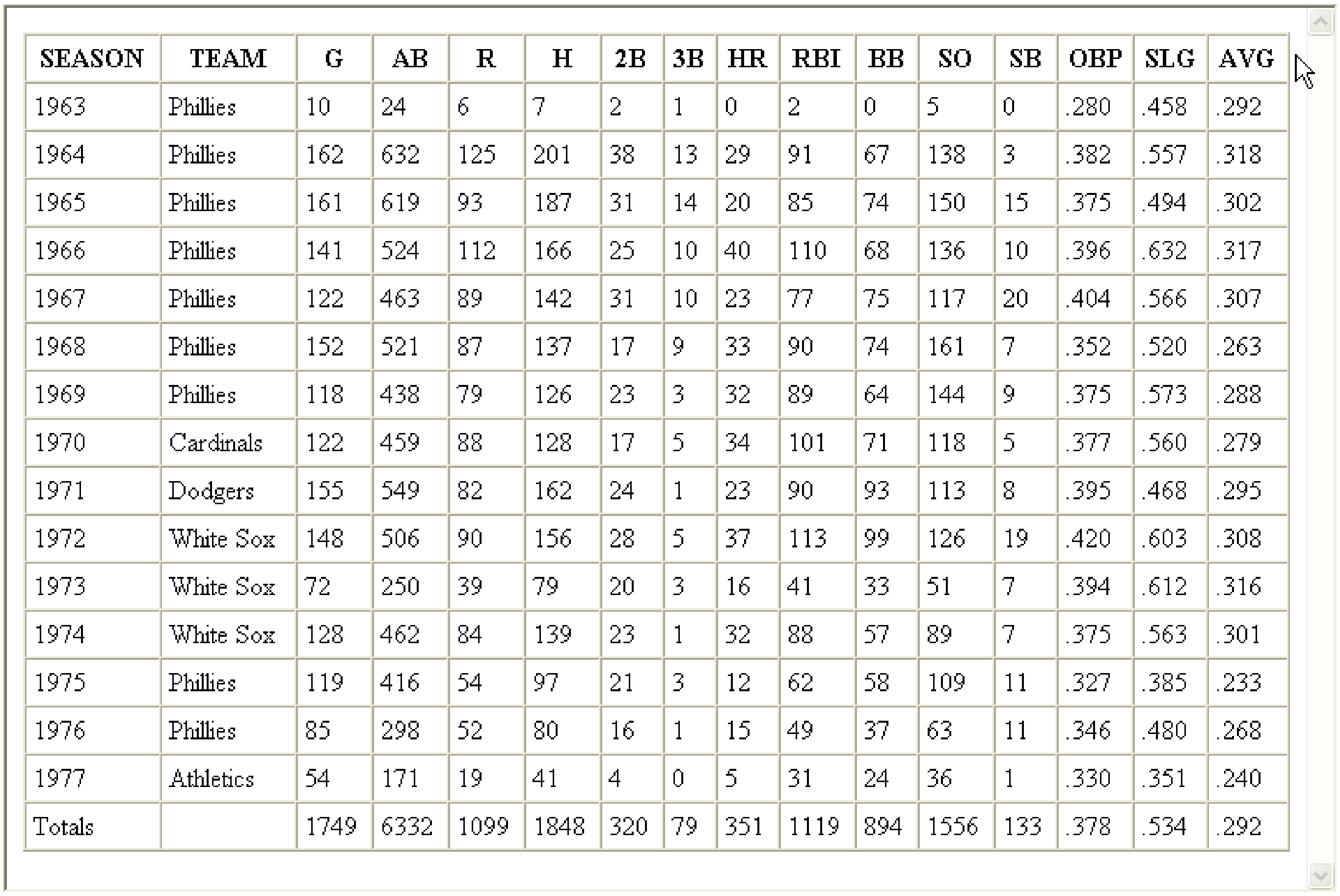
Spicing up Data Tables with Highlighting Rows

There is a time and a place to use tables in HTML. One of those times is when you have rows of data that logically fits in a table. For example, a list of employees, an order list or a inventory report. With a large table with lots of data, it can be difficult to sort through all the rows and read the correct information.

We'll build a script that will allow your table rows to "highlight" as the user mouses over the row. We'll do this using the DOM, and we'll even load the script at page load to keep things nice and clean.

The Data Table

The first thing we need is a table of data. For this we'll use one of my favourite players who played for the Chicago White Sox in the 1970's when I was a child: Dick Allen. Below is his career statistics in a completely unstyled table with a border of 1.



SEASON	TEAM	G	AB	R	H	2B	3B	HR	RBI	BB	SO	SB	OBP	SLG	AVG
1963	Phillies	10	24	6	7	2	1	0	2	0	5	0	.280	.458	.292
1964	Phillies	162	632	125	201	38	13	29	91	67	138	3	.382	.557	.318
1965	Phillies	161	619	93	187	31	14	20	85	74	150	15	.375	.494	.302
1966	Phillies	141	524	112	166	25	10	40	110	68	136	10	.396	.632	.317
1967	Phillies	122	463	89	142	31	10	23	77	75	117	20	.404	.566	.307
1968	Phillies	152	521	87	137	17	9	33	90	74	161	7	.352	.520	.263
1969	Phillies	118	438	79	126	23	3	32	89	64	144	9	.375	.573	.288
1970	Cardinals	122	459	88	128	17	5	34	101	71	118	5	.377	.560	.279
1971	Dodgers	155	549	82	162	24	1	23	90	93	113	8	.395	.468	.295
1972	White Sox	148	506	90	156	28	5	37	113	99	126	19	.420	.603	.308
1973	White Sox	72	250	39	79	20	3	16	41	33	51	7	.394	.612	.316
1974	White Sox	128	462	84	139	23	1	32	88	57	89	7	.375	.563	.301
1975	Phillies	119	416	54	97	21	3	12	62	58	109	11	.327	.385	.233
1976	Phillies	85	298	52	80	16	1	15	49	37	63	11	.346	.480	.268
1977	Athletics	54	171	19	41	4	0	5	31	24	36	1	.330	.351	.240
Totals		1749	6332	1099	1848	320	79	351	1119	894	1556	133	.378	.534	.292

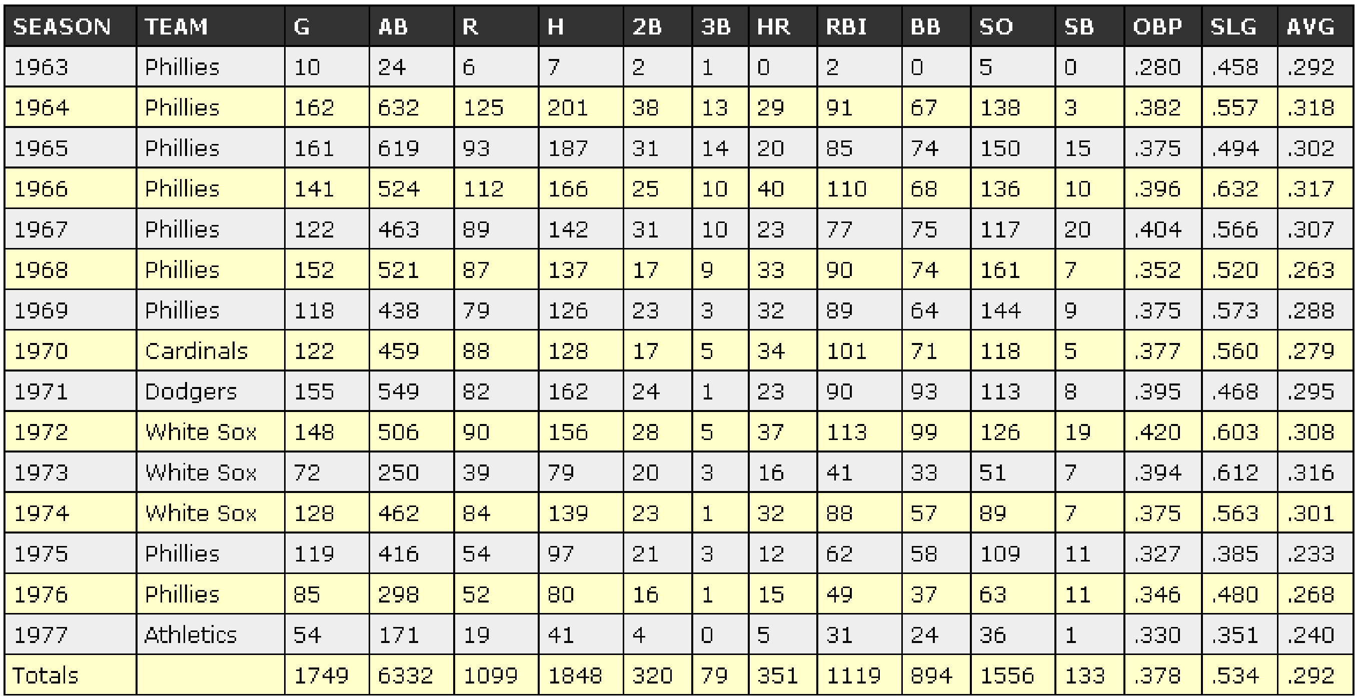
This is somewhat difficult to read, especially when you want to get a value right in the middle of the table. Adding alternating color to the rows of course will help. To quickly review, we use the following script:

```
function ColorRows(tableID)
{
     var oTable = document.getElementById(tableID);
     var oRows = oTable.getElementsByTagName("TR");
     var modRemainder;
     var newClass;

     for(var i=0; i<oRows.length; i++)
     {
           if(i == 0) continue;
           modRemainder = i % 2;
           newClass = (modRemainder == 0) ? "rowColor1" : "rowColor2";
           oRows[i].className = newClass;
     }
}
```

Which will alternate colors between the two classes **rowColor1** and **rowColor2**. In addition, we'll write an **init()** function that we can call using the body **ONLOAD** event handler to run our scripts:

```
<body onload="init();">..

--------------------------

function init()
{
     ColorRows('dickAllen');
}
```

So everything we place in the init() function will get executed when the page loads. Depending on what colors you make your row styles, you should have something like this:



SEASON	TEAM	G	AB	R	H	2B	3B	HR	RBI	BB	SO	SB	OBP	SLG	AVG
1963	Phillies	10	24	6	7	2	1	0	2	0	5	0	.280	.458	.292
1964	Phillies	162	632	125	201	38	13	29	91	67	138	3	.382	.557	.318
1965	Phillies	161	619	93	187	31	14	20	85	74	150	15	.375	.494	.302
1966	Phillies	141	524	112	166	25	10	40	110	68	136	10	.396	.632	.317
1967	Phillies	122	463	89	142	31	10	23	77	75	117	20	.404	.566	.307
1968	Phillies	152	521	87	137	17	9	33	90	74	161	7	.352	.520	.263
1969	Phillies	118	438	79	126	23	3	32	89	64	144	9	.375	.573	.288
1970	Cardinals	122	459	88	128	17	5	34	101	71	118	5	.377	.560	.279
1971	Dodgers	155	549	82	162	24	1	23	90	93	113	8	.395	.468	.295
1972	White Sox	148	506	90	156	28	5	37	113	99	126	19	.420	.603	.308
1973	White Sox	72	250	39	79	20	3	16	41	33	51	7	.394	.612	.316
1974	White Sox	128	462	84	139	23	1	32	88	57	89	7	.375	.563	.301
1975	Phillies	119	416	54	97	21	3	12	62	58	109	11	.327	.385	.233
1976	Phillies	85	298	52	80	16	1	15	49	37	63	11	.346	.480	.268
1977	Athletics	54	171	19	41	4	0	5	31	24	36	1	.330	.351	.240
Totals		1749	6332	1099	1848	320	79	351	1119	894	1556	133	.378	.534	.292

This is definitely more like it. It's easier to read the each row and find the data we need. However we can go one step further, which is the point of this script. Let's highlight the row the user mouses over.

Building the Script

We'll attach this new script to our init() function so it gets executed during the onload event of the body. You cannot run scripts on a object in a browser window that is not loaded into memory yet. The best way to do this is wait for the page to fully load, and this is what the onload event gives you.

So first we'll build the shell of our function and add it to the init():

```
function HighlightRow()
{
     // Highlight a row of data when the user mouses over
}
```

Add it to the init():

```
function init()
{
     ColorRows('dickAllen');
     HighlightRow();
}
```

In case you are wondering, "dickAllen" is the ID of our table, which is how the ColorRows() function knows which table to color. Now we have the basics in place, we can begin building the script.

Use getElementsByTagName()

We'll once again use the effective DOM method **getElementsByTagName()**. This method takes a tag name as a parameter and returns an array of that tag. You can then work with that array of tags (or objects) and do things to them.

Since we are wanting to work with table rows here, we of course will send the method "**TR**":

```
function HighlightRow()
{
     var oRows = document.getElementsByTagName("tr");
}
```

The first question that may pop into your mind at this point is: "What if I have other rows in my document that are NOT in the data table?" Good question! Glad you thought of it. The fact is, we are going to further describe the exact rows we want before we attempt to highlight them, as you will shortly see. But right now we know that we have an array of all the TRs in this document stuffed into a variable **oRows**.

Loop through rows we want

As we just mentioned, what if there are other rows than the ones in our data table? First let's set up the FOR loop:

```
function HighlightRow()
{
     var oRows = document.getElementsByTagName("tr");
     for(var i=0; i<oRows.length; i++)
     {
     }
}
```

A note about variable scope here. You might have noticed we used the variable name "oRows" in our ColorRows() function, as well as the variable "**i**" in the loop. You need to be careful about such things. If a variable is going have *local scope* only – that is only the function it is declared in can use it – then you should always preface that variable with the keyword **VAR**. This means that variable can only be used within that function, and no other functions will recognize it.

This will allow you to use "oRows" in both functions. You should be careful about using same named variables in functions. You can use any letter in your FOR loops. Its always a good idea to just include the VAR keyword in your loops (i.e., var i=0) all the time. This will save you much trouble if you have accidentally used the same letter in two loops in different functions.

Getting back to the script, we use the length property of the oRows array as our condition for continuing to loop, as we always do.
Now we have to make sure we only affect the rows we need. We'll do this with an **IF** condition, checking against the two highlight classes we are using:

```
function HighlightRow()
{
     var oRows = document.getElementsByTagName("tr");
     for(var i=0; i<oRows.length; i++)
     {
           if ((oRows[i].className == "rowColor1") || (oRows[i].className ==
"rowColor2"))
           {
           }
     }
}
```

The "||" means LOGICAL OR. If EITHER condition evaluates to true, then the IF condition evaluates to true. Notice that each comparison of class name is within its own set of brackets **(oRows[i].className == "rowColor1)** to make sure they get evaluated first, before the OR condition. It's just like you learned in math in school. If you leave comparisons ambiguous, then you are bound to get errors.

Since our ColorRows() function had applied the class rowColor1 or rowColor2 to all of the rows in our data table (except the heading which it skipped) we know we will get only those rows that we want!

Objects and properties

Remember that each row in our array is really an object. We know that each row object in our array has a class attached to it, since we just ran the ColorRows() function. "className" is the JavaScript property that returns the name of the class of an object.

To read a property of an object, we use the dot notation:

```
Object.propertyName
```

That is why when we do **oRows[i].className**, JavaScript returns the class associated with that object. The "i" of course is the index position of the FOR loop.

And now for some magic

This is where things get a little interesting, so pay close attention! What we need is a way to attach a function to an event on each row. Which events and which functions?

Well the events we are interested in are **onmouseover** and **onmouseout**. These are of course self-explanatory. Normally you would assign a function to one of these events on, say an image like this:

```
<img src="myImage.gif" onmouseover="someFunction();" onmouseout="
someFunction();" />
```

But we don't have that luxury on a table row. What we need to do is use a **function literal**.

Function literals

This has one of those "oh no, what is that..." sounding names. Although they have an intimidating name, they are fairly simple to use. What is a function literal and why do we need it? Consider this next line of code:

```
oRows[i].onmouseover =
```

Within our FOR loop we want to set each rows' onmouseover and onmouseout events to a function that will highlight our current row. Your first thought would be do to something like this:

```
oRows[i].onmouseover = HighlightCurrentRow(oRows[i]);
```

This will give you nothing but heartburn and a "not implemented" exception. For reasons we won't get into (partly because I don't fully understand them!) you cannot attach a function to an object's event handler in this fashion.

Never fear, however, because **function literals** are about to ride to the rescue! Sometimes these are also called anonymous functions as well. At any rate, a function literal looks exactly like a normal function except that it has no function name. In addition to that, we are going to place that function smack dab in the middle of our code, it isn't going to be separated out. Let's take a look:

```
function HighlightRow()
{
     var oRows = document.getElementsByTagName("tr");
     for(var i=0; i<oRows.length; i++)
     {
           if ((oRows[i].className == "rowColor1") || (oRows[i].className ==
"rowColor2"))
           {
                 oRows[i].onmouseover = function() {
                       this.className = "rowHL"; // rowHL is a new highlight
class
                 }
                 oRows[i].onmouseout = function() {
                       this.className = "rowColor1"; // one of our row colors
                 }
           }
     }
}
```

What is happening here is actually fairly logical. You are telling the current rows' onmouseover event to run a function. Then you are immediately defining that function just like you would normally. In a sense, you are assigning a no-named function to an object.

JavaScript will now be more than happy to execute this little "function literal" every time the user mouses over and out of the current row!

But what about this?

Now, you probably noticed that we did not use oRows[i].className within our function literal, but the keyword **this** instead. Why?

If you think about it in terms of variable scope, it will make sense. You've defined a new function. And even though that new function is attached to your object that you want to use, in fact on the very same line, that variable will still be out of scope. The function literal has no idea what oRows[i] is, since it was not defined within it. Nor could we do that without lots of messy code, and it wouldn't make any sense to do so.

The reason why is **this..** pun intended. Remember that the function literal is getting assigned to the events on the current row. A short example will help us understand what is happening.

Assume for the moment that we were hard coding the events into each TR tag like this:

```
<tr onmouseover="HighlightMe(this);" onmouseout="UnHighlightMe(this);">etc…
```

In the above example the keyword **this** being sent to our fictional functions would refer to the TR itself. This is exactly what is happening with our function literal. Remember that the function literal is NOT getting executed during the init() when it is called. It gets executed when the user mouses over a row.

At that point "this" refers to the current row, and the function literal sets the class accordingly.

One more problem

That's all fine and dandy, but we have one small problem. The way our function is now, we'll end up changing all rows that get moused over to only one color, but we have TWO colors. The onmouseover changes it to rowHL – which is the highlight color.

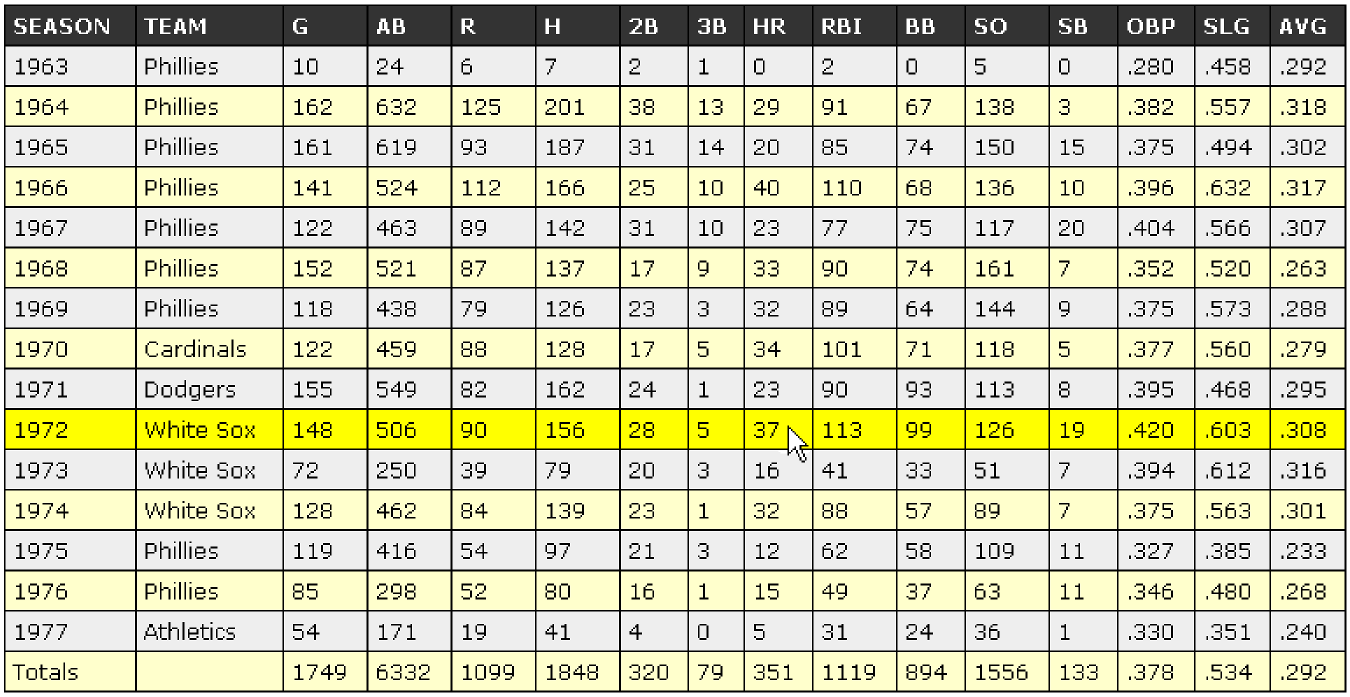
But the onmouseout can only change it back to one color the way we have it. What we need to do is somehow save the current color of the row while it gets highlighted, then use that saved value to change it back to the correct color. How can we do this?

Luckily, the **this** keyword is once again to the rescue, with a little help from an extra variable. Take a look at the function below (which incidentally is the final version):

```
function HighlightRow()
{
     var lastRowClass;
     var oRows = document.getElementsByTagName("tr");
     for(var i=0; i<oRows.length; i++)
     {
           if ((oRows[i].className == "rowColor1") || (oRows[i].className ==
"rowColor2"))
           {
                 oRows[i].onmouseover = function() {
                       lastRowClass = this.className;
                       this.className = "rowHL";
                 }
                 oRows[i].onmouseout = function() {
                       this.className = lastRowClass;
                 }
           }
     }
}
```

We've introduced a new variable called lastRowClass. This variable can be our little "save" mechanism. In the onmouseover function literal we dump the current class name into this variable BEFORE we change it to a highlight, using the **this** keyword.

Then when we change it back, we use the lastRowClass which will hold either rowColor1 or rowColor2. And what we finally get is this:



SEASON	TEAM	G	AB	R	H	2B	3B	HR	RBI	BB	SO	SB	OBP	SLG	AVG
1963	Phillies	10	24	6	7	2	1	0	2	0	5	0	.280	.458	.292
1964	Phillies	162	632	125	201	38	13	29	91	67	138	3	.382	.557	.318
1965	Phillies	161	619	93	187	31	14	20	85	74	150	15	.375	.494	.302
1966	Phillies	141	524	112	166	25	10	40	110	68	136	10	.396	.632	.317
1967	Phillies	122	463	89	142	31	10	23	77	75	117	20	.404	.566	.307
1968	Phillies	152	521	87	137	17	9	33	90	74	161	7	.352	.520	.263
1969	Phillies	118	438	79	126	23	3	32	89	64	144	9	.375	.573	.288
1970	Cardinals	122	459	88	128	17	5	34	101	71	118	5	.377	.560	.279
1971	Dodgers	155	549	82	162	24	1	23	90	93	113	8	.395	.468	.295
1972	White Sox	148	506	90	156	28	5	37	113	99	126	19	.420	.603	.308
1973	White Sox	72	250	39	79	20	3	16	41	33	51	7	.394	.612	.316
1974	White Sox	128	462	84	139	23	1	32	88	57	89	7	.375	.563	.301
1975	Phillies	119	416	54	97	21	3	12	62	58	109	11	.327	.385	.233
1976	Phillies	85	298	52	80	16	1	15	49	37	63	11	.346	.480	.268
1977	Athletics	54	171	19	41	4	0	5	31	24	36	1	.330	.351	.240
Totals		1749	6332	1099	1848	320	79	351	1119	894	1556	133	.378	.534	.292

Conclusion

Hopefully what you've seen is that using a function literal can actually save you quite a bit of time and code. Imagine if you had to attach a function to every single row for each event. If you had a table with 23 rows, that's 46 events you have to deal with. Imagine if you then had to change one!

This concept can also be used across an application. I actually used this exact code in a production application for a very large company. We had many tables of data to deal with and we found that this simply made it easier for the user to see what they wanted to see in these tables.

We also used object based thinking. You saw how the **this** keyword made things a lot easier. The more abstract you can think and the more object based you can think, the better off you will be!

Real-World JavaScript

Cookie Handling

JavaScript cookies were developed to solve an inherent problem that web pages had "back in the day." That problem was *state*. More specifically – the problem was that pages delivered via HTTP (HyperText Transfer Protocol) were state*less*.

In other words, a page lived in its own little world. It didn't know about any other pages on any other site, or even pages on its own site. It didn't know who you were the first time or the thousandth time you loaded that exact same page. Sometimes, that is just fine. In fact there are still plenty of pages we view today on the web that are stateless.

Why is Stateless a Problem?

Stateless became a problem when the web became more than research papers. Let's take the ubiquitous shopping cart as a prime example. I find a product page; I click "buy now" and add an item to my cart. How does that happen? How is the widget "added" to a cart? Furthermore, when I get to the checkout page, how does the cart know what products I've added?

Even if I am storing the data about what is in my cart with a database, I still need to know which cart to access at checkout. Take the example of personalized content; Amazon.com can make pretty good suggestions about what kind of books you might want to buy when you access their home page – if they know who you are. But how do they know who you are?

What We Need is Sessions and Persistence

In order to know who you are, a web site needs to be able to have data about you *persist*. In other words it must continue on through your time spent on the web site. It must know who you are no matter how many times you load a page or what that page is. Your time spent on a web site in this way is termed a *session*. When a web site knows who you are through multiple sessions (when you go away then come back), that is data that persists. It lasts beyond the time you spend at the site.

If the web site knows who you are during your session it can do things like present personalized or customized content and save items to a shopping cart. This data can also be used to automatically log you into a web site, remember settings and so forth.

Sometimes data is only held onto during a single session, and this is called a *temporary* session. The data is only used while you are on the site, and when you leave the data is destroyed.

What We Need Are Cookies!

Cookies can be used to store data this way – whether it is a temporary session or persistent data. JavaScript is one way cookies can be written to accomplish this. Many server-side scripting languages like PHP, JSP and ASP also provide the means to manipulate cookies. Quite often, cookies work in conjunction with a back-end database (or even some kind of data files, like an XML configuration file) to provide a customized experience to the user or to track some kind of activity like a shopping cart.

But What is a Cookie?

But how is this data saved in a cookie? What is a cookie, really? A cookie itself is not a script. We'll use a script to manipulate the cookie, but the cookie itself is nothing more than a static text file or a "passive text string" if you want to get technical.

When a user's browser allows the writing of a cookie, the data is created and stored on the user's local computer. The originating web site then can read and write its OWN cookies only, using that data as it sees fit.

When I say a site can only read its own cookies, it gets complicated very quickly. Basically, a cookie set by www.domainA.com can **never** read a cookie set by www.domainB.com **under any circumstances.** However, when we look at subdomains, it all gets tricky. Skip this next bit, if you're not using subdomains!

Let's quote Doc JavaScript, as he says it well:

"The path attribute specifies a subset of URLs in a domain for which a cookie is valid. After the domain is matched, the pathname component of the URL is compared with pathname (the value of the path attribute), and, if successful, the cookie is considered valid and is sent along with the HTTP request. The path "/foo" would match "/foobar" and "/foo/bar/html". "/" is the most general path. If a path is not specified, it defaults to the path of the document or script that set the cookie."

Further info http://www.webreference.com/js/column8/http.html

Let's take a quick look at one of these cookie files I've taken right off of my own computer:

```
B=04dcsnh00bpdb&b=2

Y=v=1&n=4qmuteejn7dtq&l=f8n4bc427/o&p=m221qp14d3000200&r=6l&lg=us&intl=us

T=z=4NpBAB4T.BABJD/qebU4gF0NjUxBk41MDI3NzJONQ--
&a=YAE&sk=DAAnxxbN7xs9R5&d=c2wBTVRJMkFUa3lOelV3TURVNU1nLS0BYQFZQUUBb2sBWlcwLQF0
aXABWUx5dkZDAXp6ATROcEJBQmdXQQ--
&af=QUFBQ0FDQUQmdHM9MTA3NDE3Mjc5MiZwcz1XU2Y4MmlMdW1mTlBtZlY2XzdmVDhnLS0-

I=ir=ca&in=4eddd0e2&i1=AAAIBJCHCQDdDjDzEzFNBJALRDSyS0S1S2TQTWViVzYQYWBhABqJCxAL
FSFTGPGRGTHyKSKnLFLJLOCzAHemeneteze0e1fMC3ABG3DDABVt
```

Now that is one ugly cookie! This is the data that was stored by a site I frequently use. This data gets changed all the time, in fact every time I access that site. Some of it is certainly a randomly generated session variable to uniquely identify me as I perform tasks on their web site.

Let's take a look at a friendlier one:

```
ASPSESSIONIDQABRRRCT=BJCPIGMBDJDLMOIDHAGOCELJ
```

Here it is a little easier to see that this site is writing a unique session using ASP technology on the server side. What we see here is a name/value pair. The name of the first piece of data is:

```
ASPSESSIONIDQABRRRCT
```

The value of this named piece of data is:

```
BJCPIGMBDJDLMOIDHAGOCELJ
```

That is what uniquely identified me on that web site.

More Cookie Facts

A cookie file cannot exceed 4k of data. That is plenty of data for anything you'll need to do with a cookie, believe me. A site may write no more than 20 cookies and you can't have more than 300 stored on your machine. If you do and you get a new one, the oldest one goes bye-bye.

Cookies can present an accessibility issue if your user does not allow JavaScript, or they disallow cookies. The general rule is that your site should at least function even if you cannot write cookies. If the user wants to use more advanced features of your web site, you'll have to simply inform them that they must accept your cookies. Yahoo! is a good example. I use Yahoo! Mail and it wouldn't work for me if I disabled cookies.

Write a Simple Cookie

So down to brass tacks: how do we implement a simple cookie? Let's get a basic scenario to use. Let's say we want to save our user's layout preference for the web site.

The property we are going to modify for the cookie is a property of the document, obviously called "cookie." So we'll be using document.cookie to read and write the cookie. Take a look at the following script:

```
function SetCookie (cookieName, cookieValue)
{
     var today = new Date();
     var expireDate = new Date(today.getTime() + 28 * 24 * 60 * 60 * 1000);
     document.cookie = cookieName + "=" + escape(cookieValue) + "; expires=" +
expireDate.toGMTString();
}
```

Hopefully this doesn't look to scary. Since we've just covered the **Date()** object you should be somewhat familiar with what is going on in the first two statements. All we are doing is setting a date that is 28 days in the future. Basically we are saying, let's have a persistent cookie that lasts 28 days (roughly a month). You could make this date really any time, even years ahead if you want. It just depends on the application and your needs. The default is the length of the current session. If no expiry is set, when the browser is closed, cookie is kaput.

As you can see our function takes two parameters, cookieName and cookieValue. They should be fairly self evident. The cookieName is of course the "name" part of a name/value pair, while the cookieValue is the "value" part. So if we wanted to set a cookie called "layout" to some value named "no right side" because our customer doesn't want to see the right sidebar (remember, personalized content) then we would call the function like so:

```
SetCookie(“layout”, “no right side”);
```

Based on that function call we want to see what happens in the line:

```
document.cookie = cookieName + "=" + escape(cookieValue) + "; expires=" +
expireDate.toGMTString();
```

Firstly notice that we are setting the document.cookie property. The rest of this statement has to do with simply building a string with concatenation. Since the name is on the left side of the name/value pair we begin with the cookieName parameter.

After the name, you must add in an "=" sign to separate the pair. The next bit needs a bit of explaining. You can see we are then placing the value of the cookie using the cookieValue parameter on the right side of the equal sign. However, it has been placed inside a method called "escape."

There are actually two related methods here (we'll see the other one when we retrieve the cookie). **escape()** and **unescape().** If you take a look at some long URLs you see on the web you will see something like this:

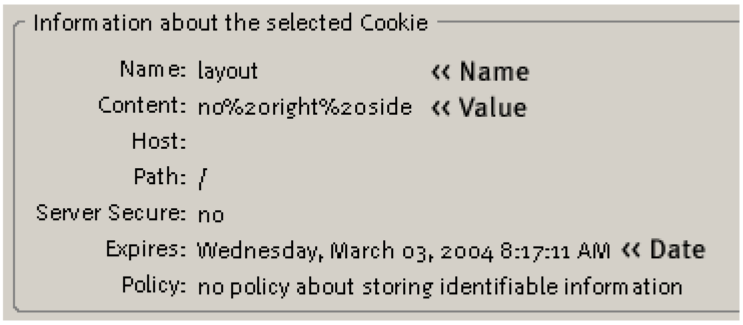
```
http://www.amazon.com/exec/obidos/subt/home/home.html/ref%3Dtab%5Fmi%5Fgw%5F1/1
04-27294070754328
```

In this string you can see some odd character with % signs, numbers and letters. This string is said to have been "escaped." In other words, characters like spaces and semi-colons (to name a couple) have been encoded so that they can be read properly by the systems that make use of the URL (like your browser and back-end systems.) Having a blank space in the URL wouldn’t do. So the escape function converts any of those odd characters you might have (we have 2 spaces in our value) to the correct escaped characters. For example, a space gets escaped to **%20**. So when our cookie is written, the value will look like:

```
no%20right%20side
```

So far the two sides of the name/value pair have been put together. Now we need to place in our date. Notice that in the next part of the string we include a semi-colon and "expires=". This tells the cookie the value is finished, and you are giving some other information about the cookie – namely the expiration date. We'll use the toGMTString() method to put the date into GMT format so it isn't simply a local time.

There you have it! A cookie has been written. If we use Firebird to run this script we can easily view what is in the cookie, and this is what we will see:



Information about the selected Cookie

Name:	layout	« Name
Content:	no%20right%20side	« Value
Host:		
Path:	/	
Server Secure:	no	
Expires:	Wednesday, March 03, 2004 8:17:11 AM	« Date
Policy:	no policy about storing identifiable information	

Notice the exact name/value pair you specified (Firebird calls the value "content" but it's the same thing). You can see the expiration date you set as well. The other information you need not worry about unless you are doing more advanced applications.

How to Retrieve the Cookie Value

Ah, but now we want to read the cookie value! When the user comes back to your site, you need to read the cookie and then show the correct display for the user. We need to know if it's "default", "no right side" or some other setting.

So we need a cookie reading function like this one:

```
function GetCookie(cookieName)
{
     var cookieNameLength = cookieName.length;
     var cookieData = document.cookie;
     var cookieLength = cookieData.length;
     var i = 0;
     var cookieEnd;

     while(i < cookieLength)
     {
           var j = i + cookieNameLength;
           if(cookieData.substring(i,j) == cookieName)
           {
                 cookieEnd = cookieData.indexOf(";",j)
                 if(cookieEnd == -1)
                 {
                       cookieEnd = cookieData.length;
                 }
                 return unescape(cookieData.substring(j+1, cookieEnd));
           }
           i++
     }
     return null;
}
```

In this case, we'll call this function by sending the cookie name ("layout") as the parameter, which we've appropriately named cookieName. Then we set up a few variables:

cookieNameLength: This is the length in characters of the name of the cookie (such as "layout")

***GOTCHA**: This length DOES NOT include the expiration date or any other part of the cookie! It is ONLY the name/value pair! Don't let this confuse you. Basically the name/value pair is the first two values in your cookie document, and that is all you can pull from your script.*

cookieData: this is just a variable that points to document.cookie
cookieLength: this is the length of the WHOLE cookie in characters
Then we set a counter as i = 0, and a cookieEnd variable which you will see how we use.

The meat of the function is really this:

```
      while(i < cookieLength)
      {
            var j = i + cookieNameLength;
            if(cookieData.substring(i,j) == cookieName)
            {
                  cookieEnd = cookieData.indexOf(";",j)
                  if(cookieEnd == -1)
                  {
                        cookieEnd = cookieData.length;
                  }
                  return unescape(cookieData.substring(j+1, cookieEnd));
            }
            i++
      }
      return null;
```

We're saying while the cookie still has characters, search through it to find out cookie name. If we go through the whole cookie and don't find it, the script will return null (or nothing). At that point, we would most likely set a default view for our user.

Inside the while loop, we start by setting a variable j to the cookie name length plus i. What this accomplishes is to allow us to use substring() to match each section of the string and see if we have a match. Since the script merely sees all the data as one long string, it has to find the cookie name *exactly* first, before it can extract the value. Remember this domain may have more than one cookie set.

So if the script ran through 3 times, the substring values in the IF condition would look like this (based on layout as the cookie name):

```
0,6
1,7
2,8
```

And so on, until it matches the cookie. So if substring(0,6) matches "layout" then we enter into the IF condition. We then use our cookieEnd variable to find the end of the cookie which is marked by a semi-colon. Remember the value is on the RIGHT side, and now that is what we are after. We need to grab the beginning and ending indexes of the value so we can extract it with substring().

```
cookieEnd = cookieData.indexOf(";",j)
```

The indexOf() method says "find out the beginning index of the character(s) I specify." And optional second parameter (j) is added as the place for the indexOf() method to START. In essence, we are starting to look at the *end* of our cookieNAME and moving on to the *end* of the string. (Since we've found the cookie name, we know the semi-colon is not in there and we can eliminate those characters in our search).

So if the script doesn't find the semi-colon from the end of the cookie name to the end of the cookie, then the semi-colon must be at the end of the string. If that is the case, this line will evaluate to true:

```
if(cookieEnd == -1)
```

-1 means "was not found." If we don't find it, we know its at the end so we set cookieEnd to the same as cookieLength – the total length of the cookie.

Now we have the two important indexes we need: j is the end of the cookieName, and cookieEnd is the end of the cookieValue. Using those two numbers we can extract the cookieValue and return it.

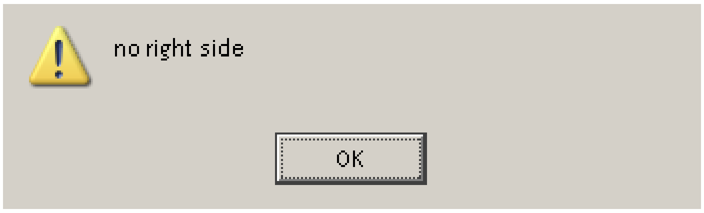
```
return unescape(cookieData.substring(j+1, cookieEnd));
```

Note how we unescape() the value, so we can actually read it – we don't want to sift through %20's and so on. The last little wrinkle is the +1 – remember the equal sign? We don’t want that returned in our value, so adding one to j skips the equal sign.

Then if you do this:

```
myCookie = GetCookie("layout")
alert(myCookie)
```

You should see:


no right side

OK

And you have your cookie value! Do with it what you will!

Conclusion

This was only a basic introduction to the cookie. There are lots more features associated with it, such as what path can access the cookie, is it secure, and issues like server-side scripting writing cookies. The issue actually can get very complex.

However, it doesn't have to be. You can use the simple functionality I've shown you above to get quite a bit done. If you are interested in reading more about cookies, here's a good resource:

http://www.cookiecentral.com

The JavaScript Date Picker

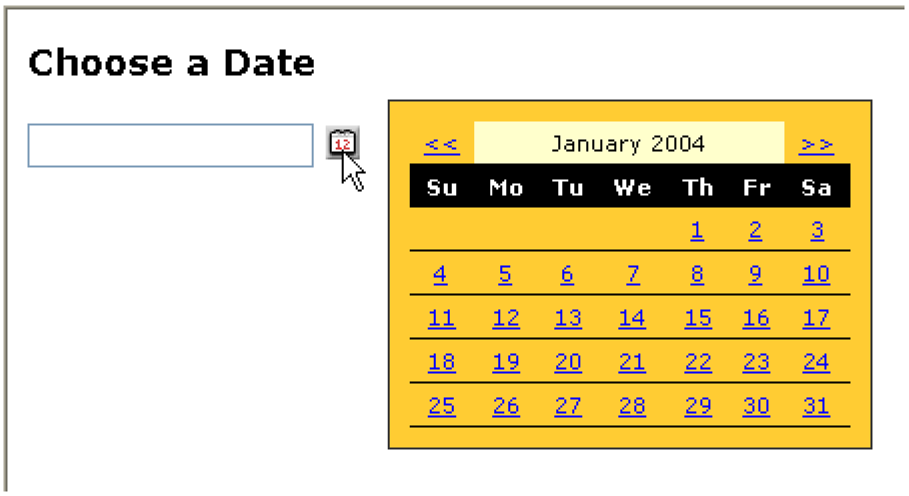
Sooner or later you're going to have the need for a user to fill in a field with a date. It's like Murphy's Law, it's inevitable. In any event, one nice way of handling the situation is to provide some kind of widget to allow the user to pick the date instead of having them type the date in manually. Typing in dates manually can often be a problem, especially with a global medium like the web; Americans write 3rd January 1/3; Europeans write 3/1. Are your dates 03/01/04? 3/1/04? 1/3/04? 1/3/2004? A date picker, when a user clicks on a date from a calendar-like interface can smooth the process.

Such a widget provides a couple of nice features. First of all, it helps you make sure the application gets good data. While it's quite easy for the user to mistype a date, a correctly built script can place a correctly formatted date in your field 100% of the time.

Secondly, it adds simple functionality to allow the user to easily select a date. The user can still type in the date if they wish, or they can make user of the widget.

How it looks

Let's first take a look at the finished product. We're going to keep this very generic, since picking a date can be used for literally thousands of applications. All we are going to assume is that we have a field where a user needs to choose a date.



Choose a Date

<<	January 2004					>>
Su	Mo	Tu	We	Th	Fr	Sa
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

How it functions

The user is prompted by the title to choose a date. There is a text field where the user could simply type in the date if they so desired. But the more obvious function is for the user to click the little calendar icon. This in turn displays the calendar to the right of the button.

We want the user to be able to navigate the months in case they want to choose a date in the future (or even the past – remember this is a generic application). Clicking on the date should place the date into the text field and hide the calendar again.

Setup: HTML and CSS

We're not going to use a popup window for the calendar. This adds complexity to the application we simply don't need, and modern browsers are much smarter than that anyway. As is our custom, we'll use a DIV tag and simply hide and show it as we have done in the past. So the HTML is very basic:

```
<h3>Choose a Date</h3>
<form name="service">
<input type="text" name="serviceDate" id="serviceDate" /> <a href="error.html"
onclick="ShowCalendar(document.forms['service'].elements['serviceDate'].value);
return false;">
<img src="calendar.gif" alt="" border="0" /></a>
</form>
<div id="calendar"></div>
```

No doubt your first question is "where are the dates of the calendar?" Actually those are written out dynamically using innerHTML each time we need our calendar. So the only elements we are going to be dealing with are the input text field, and a DIV tag where we dynamically generate the calendar each time the user needs it.

Notice that we are triggering our **ShowCalendar()** function (which obviously shows the calendar) using the **onclick** event. As always, we also add "return false" afterward to disable the link behaviour. If the user does not have JavaScript enabled, the link will then work and they will see our error page which tells them that JavaScript is required.

The only CSS we really need to see (you'll want to format the calendar as your page/ application requires) is the line that hides the calendar:

```
#calendar {
     display: none;
     position: absolute;
     top: 45px;
     left: 190px;
     border: 1px solid #333;
     background: #fc3;
     padding: 10px;
     font-size: 11px;
     font-family: verdana; }
```

The display is set to none so it does not show up on page load. Furthermore, we are using absolute positioning to place it exactly next to our button.

Setting up the script

Before we get into the main function, we need to set a couple things up. You might remember from my Date Object tutorial that JavaScript sees months as numbers. Furthermore, it numbers the months as 0-11. We can get some goofy results if we don't handle these situations. Therefore we begin by making two simple arrays:

```
var monthArray = ["January", "February", "March", "April", "May", "June",
"July", "August", "September", "October", "November", "December"];
var monthNumArray = ["01", "02", "03", "04", "05", "06", "07", "08", "09",
"10", "11", "12"];
```

By calling the index position used by the date object in these arrays, we can get back the right month we need in any situation. For example, if the month of the date object is 0, we know it is January. But we don't want a date like this:

```
00/12/2004
```

Which is what we would get, since January is 0. But using 0 as our index in the monthNumArray we can get 01 instead, which is what we need:

```
monthNumArray[0]; gives us 01/12/2004
```

The real index in JavaScript arrays runs from 0 -11, but the monthNumArray runs from 1 to 12, which is how we humans perceive the dates to be numbered.

We actually have four functions to deal with. Our main function is ShowCalendar(). That function has 3 utility functions which we will see a bit later.

The main script

So jumping right in, let's take a look at the beginning of **ShowCalendar():**

```
function ShowCalendar(dateValue)
{
     var calDiv = document.getElementById("calendar");

     // if there is no date yet, make a new one. Otherwise, use the date in the
input box
     // or the date passed by the next/previous buttons.
     var currentDate = (dateValue == null || dateValue == "") ?  new Date() :
FormatDate(dateValue) ;

     var prevMonth = new Date(currentDate);
     prevMonth.setMonth(currentDate.getMonth()-1);

     var nextMonth = new Date(currentDate);
```

```
      nextMonth.setMonth(currentDate.getMonth()+1);

      var currentYear = currentDate.getFullYear();
      var currentMonth = currentDate.getMonth();

      // shift the current day to day 1 to make calendar to build it from day 1.
      var firstDay = new Date(currentDate);
      firstDay.setDate(1);
      var currentDay = new Date(firstDay);

      // clear current calendar (if exists)
      var calHTML = "";
```

This may look a bit daunting but I will explain each line. The variable ***calDiv*** is merely a reference to the DIV where our calendar will reside.

However ***currentDate*** is an important variable, for we'll use it a lot. This variable will hold the current date object that is being used while the function works. So that is the first thing we need to establish, what is the current date?

We first have to see if a date exists already in the date field. Maybe the user selected one and realized it was wrong, or changed their mind. Once again we are using the ternary operator to do this, to avoid the longer **if…else** structures. So the condition we are checking against is in the parenthesis:

```
currentDate = (dateValue == null || dateValue == "")
```

If you haven't noticed, ***dateValue*** is the parameter being sent by the onclick event. This was done in the usual way, using the forms/elements array (**document.forms['service'].elements['serviceDate'].value** – see the HTML above).

If that value is null or blank then we simply create a new Date object by doing new Date(). If there is a value there, we have to deal with it because it is NOT a date object, it is merely a text string. You'll see this in action at the end of the script, where we take the date and format it for a text field. Plus, JavaScript has no way to "remember" that the string is a date.

So we have to make it into a date. We know what date it is because it is formatted mm/dd/yyyy. Since we know that much, all we need is a function to take those three pieces of information and create a new date object. That's where the first utility function, **FormatDate()** comes into play.

FormatDate()

```
function FormatDate(dateValue)
{
     // At this point all we have is a string like 01/01/2004
     // We must have a date object, so we make one based on the string date
     // and return it to the main function
     var dateValArray = dateValue.split("/");
     var currentDate = new Date(dateValArray[2],dateValArray[0]-
1,dateValArray[1]);
     return currentDate;
}
```

Luckily this is quite easy to do. Arrays have a wonderful method available to them called **split()**. What this does is look through a string, and when it sees the delimiter you specify (as the slash character above) it splits the string. It does this every single time it sees the delimiter. The end result is a new array with the split up values. So if the original string is 01/01/2004, then after it is split we are going to have an array with that looks like:

```
dateValArray[0] = 01
dateValArray[1] = 01
dateValArray[2] = 2004
```

So we supply the FormatDate() function with our string (*dateValue*) and use the **split()** method on it, placing it into our ***dateValArray*** variable. With this array in hand, we can create a new Date() object by sending the date constructor the month, date and year. For example, the Date() object expects the full date first, so we place the "2004" value in first. This is in the 2 position of the array.

The next value is the month – and you'll notice we subtract 1 from this value. Why do we do this? Remember the Date() object sees months as 0-11, not 1-12. We have to provide JavaScript the right data, if we don't the whole script will go wonky. JavaScript will never really have the month right, and that would cause lots of problems for us later.

Remember, we are simply taking the string date that exists in the text field and converting it into a date object so our main script can use it in the format that JavaScript can easily work with dates. It is **not** for display.

Now that we have the new date, we simply return it:

```
return currentDate;
```

Now at this point some people can get confused so let me explain something. If there is a date in the text field, we in essence did this:

```
var currentDate = FormatDate(dateValue);
```

So *currentDate* is going to hold the RESULT of FormatDate(dateValue). Well as we just saw, FormatDate(dateValue) returns a new date object that matches the string value. The variables are named the same so you hopefully get the connection. currentDate in the main function is going to accept the date object from FormatDate(), and we can then move on.

To summarize, in either case, whether there is a date string in the text field or not, we'll have a current date object to use in our function. It's important that we are always dealing with a date object as you will see.

From this point on remember that the variable *currentDate* holds a DATE OBJECT.

Next and Previous

The next couple lines deal with allowing the user to navigate the months:

```
var prevMonth = new Date(currentDate);
prevMonth.setMonth(currentDate.getMonth()-1);

var nextMonth = new Date(currentDate);
nextMonth.setMonth(currentDate.getMonth()+1);
```

Using the **setMonth()** method of the Date() object, we make some variables to hold the next and previous date. Remember, we get the date of the current month using getMonth() then either add or subtract one (this was covered in the last paragraph.)

Just a few more variables to set, then we can get to the meat.

```
var currentYear = currentDate.getFullYear();
var currentMonth = currentDate.getMonth();
```

We want to know the current year, and we want the FULL year for display purposes. We also want to keep track of the current month. In each case we use the associated method (in bold.) Obviously, **getFullYear()** gives you "2004" instead of "04."

The First Day

```
// shift the current day to day 1 to make calendar to build it from day 1.
var firstDay = new Date(currentDate);
firstDay.setDate(1);
var currentDay = new Date(firstDay);
```

The variable *firstDay*, is going to be another Date() object. We need this date so we can build our calendar from day ONE of each month. We don't want to use the *currentDate* day because it could be anything. If you start building a calendar month from 30, you won't get very far!

We can use the setDate() method to make this date the first day of the current month. The current month is of course held in our *currentDate* date object. Notice how we used that date to base our firstDay date on. When we pass a "1" as the parameter to setDate() the date becomes "1".

This can be confusing, so here is the progression (explanation in bold) – assume currentDate is equal to 02/12/2004:

```
var firstDay = new Date(currentDate); firstDay is now a date = 02/12/2004
firstDay.setDate(1); firstDay is now 02/01/2004
var currentDay = new Date(firstDay); currentDay is another new date =
02/01/2004
```

So we've taken the *currentDate* and simply moved it to the first day of that month. Then we've duplicated that date again in the variable *currentDay*. So we now have THREE date objects we're going to use:

```
currentDate: 02/12/2004
firstDay: 02/01/2004
currentDay: 02/01/2004
```

Confused? Don't be. We are now DONE with *currentDate*. It did all we needed it to do. The variable firstDay is not going to change, but we will reference it. The *currentDay* variable is going to be the actual "current day" in our calendar as we build it, and will change as such. We'll use that for comparison as well. Just keep that in the back of your mind as we continue.

Lastly we have to make sure that our variable used to write out HTML to the DIV tag is clean and empty. So we initialise that variable. Then we set the display of the calendar DIV to "block" so it shows up when we write the HTML:

```
// clear current calendar (if exists)
var calHTML = "";

// show calendar
calDiv.style.display = "block";
```

Calendar Header

The first thing we want to write out is the header of the calendar. This includes not only the month but the "previous" and "next" indicators as well. Now if any data was ever fit for a table, it's a calendar, so we'll go ahead and set it up using tables.

// write out calendar header

```
     calHTML += '<table cellpadding="4" cellspacing="0" border="0"
width="220"><tr><td>';
     calHTML += '<a href=""
onclick="ShowCalendar(\''+FormatRawDate(prevMonth)+'\'); return
false;">&lt;&lt;</a>';
     calHTML += '</td><td colspan="5">' + monthArray[currentMonth] + ' ' +
currentYear + '</td><td>';
     calHTML += '<a href=""
onclick="ShowCalendar(\''+FormatRawDate(nextMonth)+'\'); return
false;">&gt;&gt;</a></td></tr>';
     calHTML +=
'<tr><th>Su</th><th>Mo</th><th>Tu</th><th>We</th><th>Th</th><th>Fr</th><th>Sa</
th></tr>';
```

Remember the "+=" operator means "add to." If we were to only put the equals sign, we'd rewrite what was in the *calHTML* variable every time, and we don't want that. A couple important things are of notice here. First of all, notice that we switched to single quotes to encompass our HTML strings. When you are writing strings out that have to include quote marks, things get tricky in a hurry. Experience has shown that using single quotes for the main set is much easier to deal with than double (try yourself and you'll find out.) Even so, we still have to use delimiters (the slash character) when sending parameters to our function calls in the HTML string.

Remember earlier when we set variables for previous and next month? Now we get to use them. The problem is however, they are date objects. However, our ShowCalendar() function does NOT take a date object as a parameter, but a date string. If the user clicks on the next or previous arrows, we simply call our ShowCalendar() function again, sending the previous or next month as a parameter. This gets placed into our *currentDate* variable we used so much at the beginning. So if we are in February and the user clicks "next" then the *currentDate* object will be sent a March date, and the next calendar built for the user will be March, just as they expect.

But how does the "raw" date object get transformed into a date string? That's where our second utility function comes in: **FormatRawDate()**

FormatRawDate()

```
function FormatRawDate(rawDateValue)
{
     // The next/previous buttons have date object references. They need to be
strings
     // so we can send them to the function when the next/previous links are
clicked.
     // the string date is returned and placed into the link.
     var rawMonth = rawDateValue.getMonth()+1;
     var rawDay = rawDateValue.getDate();
     var rawYear = rawDateValue.getFullYear();
     var stringDate = rawMonth + "/" + rawDay + "/" + rawYear;
     return stringDate;
}
```

FormatRawDate(rawDateValue) accepts a Date() object as the parameter, which we'll call ***rawDateValue***. Since this is a date object, we can use the beautiful date methods to get what we need. So we set three simple variables for month, day and year, using the bolded methods to get the correct data.

Notice that again for the month we manipulate the value. This time we add one, since again we are getting a value of 0-11 when now we need 1-12. All that is left is to use simple concatenation to build our string, placing the slashes in the appropriate place. We then return the variable *stringDate* which is now a nice little string like "03/01/2004".

So in reality what the browser has for the user becomes this:

```
onclick=”ShowCalendar('03/01/2004');”
```

- which is just what we wanted.

The other line of note displays the current month and year:

```
calHTML += '</td><td colspan="5">' + monthArray[currentMonth] + ' ' +
currentYear + '</td><td>';
```

We already have a value for the year, but we have a number for currentMonth – we want the real month name. So we use the monthArray we made way back in the beginning to grab that correct month, using the *currentMonth* number as the index for retrieval.

Build the Calendar

Now the heart of the issue – build the calendar! One more quick variable to set up:

```
var curCell = 1;
```

We're going to need this guy or ALL our calendars will start on the first cell of the calendar, which would be Sunday of week one and we don't want that. (Accuracy, what a pain!)

We'll take this part piece by piece, and then show you the whole thing at the end.

```
// as long as we are in the current month, write out the calendar days
     while(currentDay.getMonth() == firstDay.getMonth())
     {
           // begin row
           calHTML += '<tr>';
```

Using a while loop (which says "while such and such is true – do what's inside me") we make sure we only write out dates in the current month we are in. We have a *currentDay* variable that is a date, so we get the month from that date. Remember *currentDay* is going to iterate through each day of the month.
The variable firstDay is a static variable that will NOT change. It also is a date object of which we can get the month. So if these two months are equal, then we are in the current month. Once *currentDay* kicks into the next month, the two months will NOT be equal, so the script will exit the while loop. The calendar month will be finished.

Once inside the loop we begin by writing out a table row. We then need to write out the first week of 7 days.

```
// iterate through each week
           for (var i=0; i<7; i++)
           {
                 calHTML += '<td>';
```

A week is 7 days, so we'll do a simple **for** loop that counts up to 7. Each time through 7 days we'll begin a new table CELL. Now we need to make the same check again we just made for the while loop. Because the month may end BEFORE the end of the week, we don't want to start writing out next month's values in this month's calendar.

```
// as long as this week is in the same month, write out days
     if(currentDay.getMonth() == firstDay.getMonth())
     {
```

Here is where we need to check for our first day of the month. The Date() object is great in that it is smart enough to know things like what DAY of the week the first DATE falls on. We can check that, but we don't want to write out dates until the first day starts. In other words, if January 1, 2004 starts on Thursday, we don't want to start writing "1, 2.." on Sunday, or the whole thing will be incorrect.

```
// if first day is not reached yet, write a blank
                       if(curCell <= firstDay.getDay())
                       {
                             calHTML += '&nbsp;';
                             curCell++
                       }
```

We have a ***curCell*** variable that equals 1 to start. Using the **getDay()** method of a date object, we ask JavaScript what day January 1 falls on (it happens to be Thursday.) Day in JavaScript is a number from 1-7. So January 1, 2004 starts on 5. So we want to write out blanks until we hit the 5th day of the first week.

Therefore we write out non-breaking spaces and increment *curCell*. Finally we will hit 5 and we can start writing out the days of the month.

```
// otherwise write out the date
else
{
     calHTML += '<a href="" title="Click to choose date" onclick="SelectDate('
+ currentDay.getMonth() + ',' + currentDay.getDate() + ',' +
currentDay.getFullYear() + '); return false;">' + currentDay.getDate() +
'</a>';
     curCell++
     currentDay.setDate(currentDay.getDate() + 1);
}
```

Remember we are dealing with the date object *currentDay* for each day. So we can use the methods available to the Date() object (note the methods in bold.) You can see we're introducing the last utility function we need called SelectDate(). This function will place the date string the user chooses in the text field and once again hide the calendar. We'll look at that in a moment. For now, you can simply see that we just build send the date as three separate values like '02', '12', and '2004'. The reason we don't put the slashes in here is that it creates quite a challenge to delimit the slashes with everything that is going on in that statement. It's much easier to simply add them in with the SelectDate() function.

Then in the table cell itself we put the actual date using getDate(). Here too we must increment curCell – whether we write or date or not we must do that each time a day passes.

Lastly (and finally) we actually increment the day to the next using the setDate() method by adding 1.

Closing it up

Once the month is over, we close things up:

```
}
calHTML += '</td>';
     }
calHTML += '</tr>';
}
```

After each day we close the table cell, after each week we close the table row. Once we exit the while loop and the month is over, we also need to close the table and finally, send that HTML string we just built to our DIV tag:

```
     calHTML += '</table>';
     calDiv.innerHTML = calHTML;
}
```

The SelectDate() Function

```
function SelectDate(selMo, selDay, selYear)
{
     var dateField = document.getElementById("serviceDate");
     var calDiv = document.getElementById("calendar");
     dateField.value = monthNumArray[selMo] + "/" + selDay + "/" + selYear;
     calDiv.style.display = "none";
}
```

The function takes those three values we set while writing out the calendar. We tell the function which element to write to (*dateField*), that's our text field. We also tell the function about the DIV we want to write to (*calDiv*.) Then we both build our date by adding the slashes and apply that string to our text field using the **value** property (the line in bold).

The user has selected a date, so we now hide the display of the calendar, and we're done! The date is in the field.

Full Script Listing

Here's the full script, with all 3 utility scripts as well so you can copy and paste it at will:

```
var monthArray = ["January", "February", "March", "April", "May", "June",
"July", "August", "September", "October", "November", "December"];
var monthNumArray = ["01", "02", "03", "04", "05", "06", "07", "08", "09",
"10", "11", "12"];

function SelectDate(selMo, selDay, selYear)
{
     var dateField = document.getElementById("serviceDate");
     var calDiv = document.getElementById("calendar");
     dateField.value = monthNumArray[selMo] + "/" + selDay + "/" + selYear;
     calDiv.style.display = "none";
}

function FormatDate(dateValue)
{
     // At this point all we have is a string like 01/01/2004
     // We must have a date object, so we make one based on the string date
     // and return it to the main function
     var dateValArray = dateValue.split("/");
     var currentDate = new Date(dateValArray[2],dateValArray[0]-
1,dateValArray[1]);
     return currentDate;
}

function FormatRawDate(rawDateValue)
{
     // The next/previous buttons have date object references. They need to be
strings
     // so we can send them to the function when the next/previous links are
clicked.
     // the string date is returned and placed into the link.
     var rawMonth = rawDateValue.getMonth()+1;
     var rawDay = rawDateValue.getDate();
     var rawYear = rawDateValue.getFullYear();
     var stringDate = rawMonth + "/" + rawDay + "/" + rawYear;
     return stringDate;
}

function ShowCalendar(dateValue)
{
     var calDiv = document.getElementById("calendar");

     // if there is no date yet, make a new one. Otherwise, use the date in the
input box
     // or the date passed by the next/previous buttons.
     var currentDate = (dateValue == null || dateValue == "") ?  new Date() :
FormatDate(dateValue) ;

     var prevMonth = new Date(currentDate);
     prevMonth.setMonth(currentDate.getMonth()-1);

     var nextMonth = new Date(currentDate);
     nextMonth.setMonth(currentDate.getMonth()+1);

     var currentYear = currentDate.getFullYear();
     var currentMonth = currentDate.getMonth();

     // shift the current day to day 1 to make calendar to build it from day 1.
     var firstDay = new Date(currentDate);
     firstDay.setDate(1);
     var currentDay = new Date(firstDay);

     // clear current calendar (if exists)
     var calHTML = "";

     // show calendar
     calDiv.style.display = "block";

     // write out calendar header
     calHTML += '<table cellpadding="4" cellspacing="0" border="0"
width="220"><tr><td>';
     calHTML += '<a href=""
onclick="ShowCalendar(\''+FormatRawDate(prevMonth)+'\'); return
false;">&lt;&lt;</a>';
     calHTML += '</td><td colspan="5" class="calhead">' +
monthArray[currentMonth] + ' ' + currentYear + '</td><td>';
     calHTML += '<a href=""
onclick="ShowCalendar(\''+FormatRawDate(nextMonth)+'\'); return
false;">&gt;&gt;</a></td></tr>';
     calHTML +=
'<tr><th>Su</th><th>Mo</th><th>Tu</th><th>We</th><th>Th</th><th>Fr</th><th>Sa</
th></tr>';

     var curCell = 1;

     // as long as we are in the current month, write out the calendar days
     while(currentDay.getMonth() == firstDay.getMonth())
     {
           // begin row
           calHTML += '<tr>';
           // iterate through each week
           for (var i=0; i<7; i++)
           {
                 calHTML += '<td>';

                 // as long as this week is in the same month, write out days
                 if(currentDay.getMonth() == firstDay.getMonth())
```

```
                  {
                        // if first day is not reached yet, write a blank
                        if(curCell <= firstDay.getDay())
                        {
                              calHTML += '&nbsp;';
                              curCell++
                        }
                        // otherwise write out the date
                        else
                        {
                              calHTML += '<a href="" title="Click to choose date"
onclick="SelectDate(' + currentDay.getMonth() + ',' + currentDay.getDate() +
',' + currentDay.getFullYear() + '); return false;">' + currentDay.getDate() +
'</a>';
                              curCell++
                              currentDay.setDate(currentDay.getDate() + 1);
                        }
                  }
                  calHTML += '</td>';
            }
            calHTML += '</tr>';
     }
     calHTML += '</table>';
     calDiv.innerHTML = calHTML;
}
```

Conclusion

As you can see working with the Date () object is both powerful and confusing! But we can be glad it offers the methods it does or things would be much more difficult. What makes this script more complex is writing out a complicated string of HTML, while keeping track of what month, week and day you are in.

The best thing to do with complex scripts like this is to try and understand one piece at a time. While you might have trouble taking in the thing as a whole, grabbing it section by section is much easier.

It also helps to comment things out and see what breaks. Play around with it. Add some alerts and see what values come up. Notice that the day isn't formatted with a "0" before it if it is less than 10 – see if you can add that in!

The JavaScript News Ticker

When the web was born, web authors inevitably put up some pretty cheesy web sites. When JavaScript was born, cheese rose to heights unimagined! The news ticker is one of those things of which I'm not sure belongs in the cheese category or not. It can be a useful little tool to draw attention to the latest headlines.

One such ticker can be seen at the BBC News web site, over at http://news.bbc.co.uk/. I think the nice thing about the ticker is that it is done in a tasteful manner. It doesn't dominate the page. Each news piece is a link to that story, and it doesn't use any real cheesy effects to draw undue attention to it.

Each title stays around long enough so it doesn't get annoying. With these things in mind, it seems like it could be a nice addition to a web site.

So let's tear it apart, build it back up again in our patented simplified manner and make use of it! In the meantime, we'll even make use of a two dimensional array.

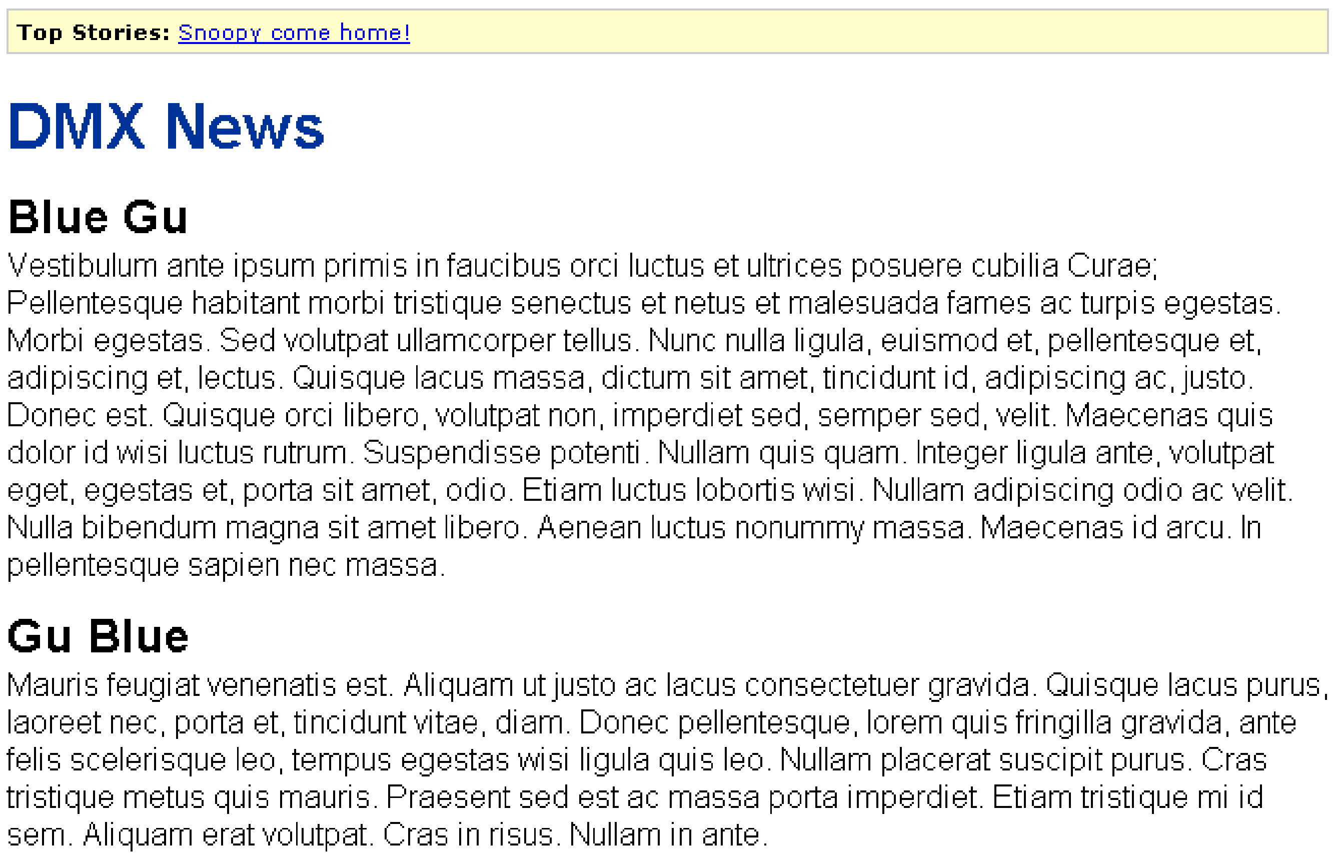
Ticker Features

The ticker doesn't have a lot going on, and as mentioned that is what we like about it. Basically it's a group of news items behind the scenes that are stored in an array, along with the associated links. The script will write out each character with a short delay, giving it the "ticker" effect using innerHTML.

As soon as it has begun being written, the title is a link right away. And each news title will stay around long enough to be read and then clicked.

The HTML

The HTML for the ticker is pretty simple. What I've done here is mocked up the simplest page possible so you could see the ticker in some form of context. So we'll have the ticker at the top of the page, the web site name, and two pretend stories on our "home page." (Certainly you're going to want to do a better design job than this!)



Top Stories: Snoopy come home!

DMX News

Blue Gu

Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Morbi egestas. Sed volutpat ullamcorper tellus. Nunc nulla ligula, euismod et, pellentesque et, adipiscing et, lectus. Quisque lacus massa, dictum sit amet, tincidunt id, adipiscing ac, justo. Donec est. Quisque orci libero, volutpat non, imperdiet sed, semper sed, velit. Maecenas quis dolor id wisi luctus rutrum. Suspendisse potenti. Nullam quis quam. Integer ligula ante, volutpat eget, egestas et, porta sit amet, odio. Etiam luctus lobortis wisi. Nullam adipiscing odio ac velit. Nulla bibendum magna sit amet libero. Aenean luctus nonummy massa. Maecenas id arcu. In pellentesque sapien nec massa.

Gu Blue

Mauris feugiat venenatis est. Aliquam ut justo ac lacus consectetuer gravida. Quisque lacus purus, laoreet nec, porta et, tincidunt vitae, diam. Donec pellentesque, lorem quis fringilla gravida, ante felis scelerisque leo, tempus egestas wisi ligula quis leo. Nullam placerat suscipit purus. Cras tristique metus quis mauris. Praesent sed est ac massa porta imperdiet. Etiam tristique mi id sem. Aliquam erat volutpat. Cras in risus. Nullam in ante.

The code for the above is:

```
<div class="tickerDiv"><span class="stories">Top Stories: </span><a
id="tickerHREF" href=""></a></div>

<h1>DMX News</h1>

<h2>Blue Gu</h2>
<p>Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere
cubilia Curae; Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Morbi egestas. Sed volutpat ullamcorper
tellus. Nunc nulla ligula, euismod et, pellentesque et, adipiscing et, lectus.
Quisque lacus massa, dictum sit amet, tincidunt id, adipiscing ac, justo. Donec
est. Quisque orci libero, volutpat non, imperdiet sed, semper sed, velit.
Maecenas quis dolor id wisi luctus rutrum. Suspendisse potenti. Nullam quis
quam. Integer ligula ante, volutpat eget, egestas et, porta sit amet, odio.
Etiam luctus lobortis wisi. Nullam adipiscing odio ac velit. Nulla bibendum
magna sit amet libero. Aenean luctus nonummy massa. Maecenas id arcu. In
pellentesque sapien nec massa.</p>

<h2>Gu Blue</h2>
<p>Mauris feugiat venenatis est. Aliquam ut justo ac lacus consectetuer
gravida. Quisque lacus purus, laoreet nec, porta et, tincidunt vitae, diam.
Donec pellentesque, lorem quis fringilla gravida, ante felis scelerisque leo,
tempus egestas wisi ligula quis leo. Nullam placerat suscipit purus. Cras
tristique metus quis mauris. Praesent sed est ac massa porta imperdiet. Etiam
tristique mi id sem. Aliquam erat volutpat. Cras in risus. Nullam in ante. </p>
```

Past that into your document body.

What's different?

We've made some significant changes to the BBC script of which this script is based on. For starters, the "Top Stories" descriptor really shouldn't be part of the link. It's more readable when it isn't, and moreover, we don't have to deal with it programmatically this way either.

We've used our two dimensional array technique as you will see instead of two separate arrays. Really in this case, neither way has a real advantage over the other, but it's nice to show a real world example.

The BBC ticker used a "widget" character at the end of the string as it was writing out the news item. You may not even notice it unless you look closely. I felt this was unnecessary, again something you have to deal with programmatically that doesn't add much (if anything at all) to the usability of the feature. So that was removed.

Then the whole script was simplified and re-written from scratch.

Where do the stories come from?

As we'll see shortly, our arrays hold the data for the news item titles and HREFs for the links. The question is where do these come from? There are plenty of ways on the server that this information could be inserted into the script. For our purposes for this week, we'll assume they are placed there by some content manager using some kind of mysterious web tool! (Adds a bit of mystery, no?)

Setup the script

To set up the script we need to set some default values and populate the array with values:

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

// ticker setup
var charDelay = 50;
var storyDelay = 5000;
var numStories = 4;

// two dimensional array to hold stories and links
var storyMatrix = new Array();
storyMatrix[0] = new Array();
storyMatrix[1] = new Array();

// column 1: titles
storyMatrix[0][0] = "Holy JavaScript, Batman!";
storyMatrix[0][1] = "Leaps tall buildings in a single bound!";
storyMatrix[0][2] = "Your friendly neighborhood Spiderman!";
storyMatrix[0][3] = "Snoopy come home!";

// column 2: links
storyMatrix[1][0] = "http://www.batman.com/";
storyMatrix[1][1] = "http://www.supermanhomepage.com/";
storyMatrix[1][2] = "http://www.marvel.com/minisites/index.htm?family=spider-
man";
storyMatrix[1][3] = "http://www.snoopy.com";
```

The first thing we have is three global variables. The first two, charDelay and storyDelay are values in milliseconds that will be used as delays. If we simply wrote out the title character by character without setting a delay between each one, it would go by so fast the human eye would not see it "tick" as it were. So charDelay will put 50 milliseconds between each character in the title string.

Then storyDelay will put 5000 milliseconds (or 5 seconds) between the end of the first title and the start of the next. This will give the reader enough time to read the title and decide to click on it. If you think this is too little (or too much) time, you can adjust it to your liking.

Next we have the storyMatrix array. Just note that the first array (column) holds the actual news title, and the second array (column) holds the URL of which to link to. Now that we have these things arranged, we can move on to the functions.

InitTicker() and NewsTicker()

The ticker has two associated functions. The first, InitTicker() simply starts the process when the page loads. The NewsTicker() recursively (that means it calls itself) runs over and over, updating the titles and HREFs.

InitTicker()

This function is simple and to the point:

```
function InitTicker()
{
     // Default values
     currentStory     = -1;
     currentLength    = 0;

     //The top stories anchor tag
     oAnchor = document.getElementById("tickerHREF");
     NewsTicker();
}
```

Our variables currentStory and currentLength will be used to let us know which story we are on, and what part of the title string we are on. The only other thing we need to take care of is letting our script know the HTML element we will write the news item to, our anchor tag. We use getElementById() to do this. Note, as usual we're not going to support legacy browsers with this script.

NOTE: We aren't using the VAR keyword for these variables in the init function, because they are also global variables. Our NewsTicker() function needs to see these variables too!

This point is crucial to do in this order. If we try and run the ticker too soon before the NewsTicker() function knows where to act, our script will error out. Lastly, we call the main function, NewsTicker().

Our init function is attached to the onload event:

```
window.onload = InitTicker;
```

Again note the absence of the brackets. This tells our event not to evaluate the function until the event has fired. Since we aren't sending any parameters to our function, we don't need to go though the anonymous function mess we had to deal with earlier.

Now when the page is loaded, our main function will begin running.

NewsTicker()

This is the main function, which calls itself (recurses) over and over. Basically we run it and leave it alone. We don't care how many times the ticker refreshes itself. Frankly, on this type of a site we don't expect the user to spend a whole lot of time on the home page anyway, as the front page of a news site is a portal to the main stories which it links to. This allows us to "set it and forget it!" (Pardon the late-night infomercial reference.)

```
function NewsTicker()
{
     var delay;

     // Get next story if title is done from previous
     if(currentLength == 0) // length of title being written out
     {
           currentStory++;
           currentStory = currentStory % numStories;

           // use HTML entity for quotes so we don't mess up the anchor
           currentTitle = storyMatrix[0][currentStory].replace(/&quot;/g,'"');

           oAnchor.href = storyMatrix[1][currentStory];
     }

     // Write title to anchor
     oAnchor.innerHTML = currentTitle.substring(0, currentLength);

     // adjust length of substring and set delays
     if(currentLength != currentTitle.length)
     {
           currentLength++;
           delay = charDelay;
     }
     else
     {
           currentLength = 0;
           delay = storyDelay;
     }

     // Recurse ticker
     setTimeout("NewsTicker()", delay);
}
```

Let's step through and see what is happening.

```
var delay;
```

We're going to use this variable to hold both of our delays – the character and story delay as you will see.

```
	if(currentLength == 0) // length of title being written out
	{
		currentStory++;
		currentStory = currentStory % numStories;

		// use HTML entity for quotes so we don't mess up the anchor
		currentTitle = storyMatrix[0][currentStory].replace(/&quot;/g,'"');

		oAnchor.href = storyMatrix[1][currentStory];
	}
```

This IF loop first checks to see if we are at the beginning of a new story. The currentLength will be zero at the beginning of a new story (or the first run of a script as set by the init function.) Of course right away currentLength IS equal to 0. So we'll jump inside the loop.

We first iterate currentStory by one (remember var++ is shorthand for variable = variable + 1). So it goes from -1 to 0. Then we'll do something that seems possibly complicated. We'll use the modulo – which is spiffy smart talk for "remainder" to set the story number. I'm no math expert myself, but the concept is pretty simple. You divide variableA by variableB and the *remaining* number is the modulo (note the % sign is the modulo operator). The best way to really understand this is to use some alerts and see what is really going on like so:

```
alert(currentStory % numStories);
```

You'll see that you always end up with a number of 0-3, the remainder of the division between the two numbers. This number is important because it of course ties into our arrays – 0 – 3 tells us which title and HREF to write out.

The next line shows this in action and throws a little regular expression in for good measure:

```
currentTitle = storyMatrix[0][currentStory].replace(/&quot;/g,'"');
```

We know that the 0 column of the storyMatrix array holds our titles. The first time through we will end up setting storyMatrix[0][0] which equals "Holy JavaScript, Batman!" (look at the matrix above if you are unclear).

Tacked on to this is a nice little method called **replace().** Replace can take a regular expression and a string to replace as its parameters. Below is the format:

```
String.replace(regex, stringToReplace);
```

Our regular expression in this case is:

```
/&quot;/g
```

The g stands for global, and we all recognize the **&quot;** entity. This means wherever a match is made, the entity will be placed there (or replaced of course) instead of what was matched. Of course what we are matching here is quotes, as you see in the second parameter:

```
'"'
```

It can be tricky to see because the DOUBLE quote is quoted by SINGLE quotes! Just note that we are searching for double quotes, and replacing them with the character entity &quot;. Why? Because if we don't then our news title string can get all messed up and break literally our whole page! We don't want that.

Having safely taken care of that, we place the actual URL in the HREF of our anchor tag:

```
oAnchor.href = storyMatrix[1][currentStory];
```

This time the [1] column of our matrix points to the HREFs in our array, and again currentStory is 0, since this is the first time through.

Actually writing the title

Now that we have the title and HREF solved, we can begin actually writing characters.

```
oAnchor.innerHTML = currentTitle.substring(0, currentLength);
```

Remember substring() tells you the beginning and ending positions in the string. We of course always want to start at the beginning, and end at the currentLength – or current character. As you might have just guessed we really re-write out the whole string each time, adding one more letter. Remember innerHTML wipes the WHOLE HTML string inside and replaces it.

This may seem wasteful on the surface, however you can see how smoothly the script runs. Furthermore, it's a very low overhead to write it out, and it would actually be more work to try and keep what was written there when we can just redo the whole thing. Oddly, sometimes scripting works this way.

```
      // adjust length of substring and set delays
      if(currentLength != currentTitle.length)
      {
            currentLength++;
            delay = charDelay;
      }
      else
      {
            currentLength = 0;
            delay = storyDelay;
      }
```

We now enter an IF…ELSE control that checks to see if we have reached the end of our news title. If our currentLength (the character position we are at) is NOT equal (!=) to the length of the currentTitle we are on, we know we still have characters to write out.

Therefore we enter into the IF clause. We increment again, this time the currentLength variable to move on to the next character. We then set our delay variable we talked about earlier to the charDelay (which was 50 milliseconds). We'll use that in just a moment.

Had we been at the end of the title – and the currentLength equalled the currentTitle.length then we would have gone into the ELSE clause. We reset the currentLength and use the storyDelay (5 seconds).

Since currentLength is again 0, we'll re-enter our IF condition at the top and the story will get bumped to the next title. The whole process begins again.

The last part is to recurse the function and use the proper delay.

```
 setTimeout("NewsTicker()", delay);
```

We've talked about setTimeout before. Basically it states run stated function (the first parameter) after waiting a specified time (the second parameter, in this case 'delay'). You have to quote the function call or it will be evaluated immediately, which you don't want.

Notice how we used the variable delay for both the character and story delay so we didn't have to write twice the amount of code.
That's it! Load the page and see your wondrous ticker!

Conclusion

Play around with the delays to get the effects you want. You'll realize that this ticker could really be placed anywhere you'd like on your site. And you could use it for other things other than news. Just make sure it's not too cheesy!

DMXzone

The History of DMXzone

DMXzone was founded in Feb 2001 by **George Petrov**. It was then called UDzone after the Macromedia product UltraDev that preceded Dreamweaver MX. By April 2001 we'd already been asked by Macromedia to speak at the Macromedia UCON 2001 conference in New York. Since then, we've grown to over 150,000 registered members of all levels and locations, who come together to share knowledge and learn from each other. We are an independent community and are in no way connected with Macromedia, the makers of Dreamweaver MX.

In May 2003, we launched our very successful Premium Tutorials track, publishing professionally written tutorials by a team of authors for an affordable price every day, as we ourselves were tired of shelling out lots of money for computer books full of redundancy and newbie's explanation. This premium track runs alongside the free content submitted by members.

What do we do

Membership of the community is free. You can view most content on the site without registering, but when you become a member you can add your own paragraphs, tutorials, news items, extensions, opinion polls and participate in the forums. To purchase extensions or download free extensions, you need to become a member.

The DMXzone Team and Manager Team consists of professionals and volunteers who work hard to bring you the extensions that you are asking for, give you the support that you need when you have questions and to bring you the latest information pertaining to web development. We like to encourage our visitors to actively participate, that is why we organize competitions, run opinion polls, let you rate articles, extensions and tutorials and let you add your own articles.

Javascript for breakfast

Crunching scripts for your coffee table

Let's face the facts. JavaScript is an essential tool to have in your toolbox. Even if you create your HTML pages with an application that creates scripts for you, there will come a time when you need a script that can't be automatically generated. There also might come a time when you want to modify one of those scripts. Most importantly, companies look to hire developers with a diverse toolset. In today's challenging job market, this can be essential to your success.

This book is for anyone with an interest in developing their JavaScript skills, the book uses very clear examples that enable you to master the programming language. It's also a useful reference for developers.

Tom Dell'Aringa is our JavaScript guru; he wants to teach you how to correctly use JavaScript by giving the basics of scripting first, then covering some simple, commonly used scripting next. Then he will move on with topics like always making sure that the code is efficient and that the user experience is good.

During the early 1990's, Tom spent his days as a graphic designer for a small publishing house. His boss threw a World Wide Web book at him one day, "encouraging" him to research it. Tom fell in love and quickly made the transition from print to web technologies, teaching himself along the way. During the wild ride of the dot.com era, Tom worked for a start up, web integrator Scient and a failed small design firm that bounced his paychecks. Forever fascinated by Peanuts comic strips, animation and history, he recently completed his first mini 3d film. He holds a B.A. in Fine Art from Columbia College, Chicago.

Javascript for breakfast