builder au
by developers for developers

Tap into advanced JavaScript functions

Edmond Woychowsky - 2003/01/03 08:50:00

http://www.builderau.com.au/webdev/scripting/soa/Tap-into-advanced-JavaScript-functions/0,339024692,320270945,00.htm

JavaScript's advanced functions will add power and punch to your Web site. Here's how to leverage recursion and passing by reference and value.

JavaScript functions are a convenient way to package sequences of instructions designed to perform specific tasks or to act as constructors for classes. You can find oodles of documentation on the Internet about using basic functions, but just try to find some info on advanced features of JavaScript functions. So I hope to provide some insight on leveraging recursion and other high-end capabilities.

Before diving into the harder stuff, let's look at the ways that you can define functions in JavaScript.

Defining JavaScript functions
JavaScript provides three ways to define functions:

- The function statement
- The *Function()* constructor
- The function literal

Let's create a function called *add* to demonstrate these three approaches. The purpose of this function is to take the two passed arguments, add the arguments, and return the result.

Listing A (all listings are at the bottom of this page) displays how to create a function using the function statement, the most common tactic. **Listing B** illustrates function creation using the *Function()* constructor. I use the *Function()* constructor for event handlers, like *onChange* or *onClick*. The advantage of using the *Function()* constructor is that it provides the ability to use the *this* keyword. The *this* keyword is a reference to the object through which the function is invoked; it gives you the ability to pass the object to a function. **Listing C** shows how to define a function as a literal, and also illustrate that JavaScript functions are strings. This means that the code:

xyzzy = add; alert(xyzzy(2,2))

will produce the same results as:

alert(add(2,2))

One advantage of defining functions as literals is that it lets you copy server-side JavaScript functions to the client side.

Now that we've hashed out the basics, let's take a look at some more advanced applications of JavaScript.

JavaScript functions as class constructors
JavaScript functions can act as class constructors to create custom objects. Because instances of these classes are created using the new operator, they can use the *this* keyword to reference the new object, as does the *Function()* constructor shown in Listing B. The use of the *this* keyword in class constructors allows instances of classes to have both methods and properties, as shown in **Listing D**.

Code Listings

Listing A

```
function add(x,y) {
     return x + y;
}
```

Listing B

```
var add = new Function('x','y','return x + y;');
```

Listing C

```
var add = function(x,y) {return x + y;};
```

Listing D

```
function myMath() {
                                //    Properties
     this.result;          //    Result of method

                                //    Methods
     this.add = myAdd;           //    Add method

     function myAdd(x,y) {
          this.result = x + y;

          return this.result;
     }
}

var math = new myMath();        //    Create an instance

alert(math.add(2,2));
alert(math.result);
```

Functions with a variable number of arguments
Each of the function examples we've discussed so far has had a fixed number of arguments that were referenced by using the argument's name. While this works, the problem is that often the number of arguments can be varied. Using the *add* function from Listing A, the result of *add(2,2)* is the same as *add(2,2,2)*, which can lead to unexpected results. You can handle a variable number of arguments to a function through the arguments object, which accomplishes this task through the length property and the arguments collection, as shown in **Listing E**.

The arguments object also provides the *callee* and the *caller* properties. The *callee* property is a reference to the current JavaScript function. The *caller* property is a reference to the function that called the current function. In addition, it can be used to access the calling function's arguments in the same manner as the arguments object does for the current function. The reference to the calling functions argument length is arguments.caller.length, while argument.caller[0] references the first argument in the collection.

When dealing with arguments in JavaScript, it's important to remember that with the exception of objects that are defined using the new operator, all arguments are passed by value. So, excluding object, any changes made to an argument will be gone when processing returns to the calling function. **Listing F** illustrates the difference between passing arguments by value and passing arguments by reference.

Recursion
I have a button that says "Recursion – adj. see recursion," which pretty much sums up the meaning of recursion. Recursion happens when a function either directly or indirectly calls itself. Recursion can be dangerous, because care must be taken to provide an exit to the recursion, but it is a useful way to handle some problems. An example of this is the mathematical function factorial (!), which is the product of all positive integers from 1 to any given number. So, 5! is equal to 120 and 6! is equal to 720, and so on. **Listing G** shows a recursive implementation of factorial.

Recursion can also be performed in a class's method, but it can be a little dicey. Attempting recursion by calling the class's method name will result in a JavaScript "Object Expected" error. To avoid this error, the method's underlying function name or arguments.callee must be used to perform recursion. Using Listing D as a starting point, I added the factorial method to the myMath class, as shown in **Listing H**

Conclusion
Since functions provide an integral part of JavaScript, the basics are well documented. Unfortunately, documentation outlining some of the more advanced features of JavaScript functions is virtually nonexistent. While the examples provided here do not by any means cover the complete ins and outs of advanced JavaScript functions, they do serve as a reference point from which to move forward.

Code Listings

Listing E

```
function add(x,y) {
     var fltResult = 0;

     for(var i=0;i < arguments.length;i++)
          fltResult += arguments[i];
```

```
    return fltResult;
}
```

Listing F

```
var x = 2;
 var y = new Array();
 y.push(3);
 document.writeln('initialize: x = ' + x + '
');
 byValue(x);
 document.writeln('initialize: x = ' + x + '
');
 document.writeln('initialize: y[0] = ' + y[0] + '
');
 byReference(y);
 document.writeln('initialize: y[0] = ' + y[0] + '
');
 function byValue(x) {       x += 2;       document.writeln('byValue: x = ' + x + '
');
 }
 function byReference(y) {       y[0] += 3;       document.writeln('byValue: y[0] = ' + y[0] + '
');
 }
```

Listing G

```
function factorial(x) {
    if(x < 2)
        return 1;
    else
        return x * factorial(x - 1);
}
```

Listing H

```
function myMath() {
//Ã,Â Properties
this.result;Ã,Â //Ã,Â Result of method

//Ã,Â Methods
this.add = myAdd;Ã,Â //Ã,Â Add method
this.factorial = myFactorial;Ã,Â //Ã,Â Factorial method

function myAdd(x,y) {
this.result = x + y;

return this.result;
}

function myFactorial(x) {
if(x < 2)
return 1;
else {
this.result = x * arguments.callee(x - 1);

return this.result;
}
}
}

var math = new myMath();Ã,Â //Ã,Â Create an instance

for(var i=0;i <= 10;i++)
document.write(i + '! = ' + math.factorial(i) + '
');
```