Interactive Web Development

Functions

Dr Russ Ross

Dixie State University—Computer and Information Technologies

Spring 2016

Functions

Reading: *JavaScript: The Good Parts*, Chapter 4

Functions

The implementation of functions is one of the best parts of JavaScript. A function encloses a set of statements:

- Functions are the fundamental modular unit of JavaScript
- Functions are used for code reuse, information hiding, and composition
- Functions are used to specify the behavior of objects

Function objects

In JavaScript, functions are objects:

- Objects are collections of name/value pairs having a hidden link to a prototype object
- Objects produced from object literals are linked to `Object.prototype`

Function objects are also collections of name/value pairs:

- Function objects are linked to `Function.prototype` (which is itself linked to `Object.prototype`)
- Every function is created with two additional hidden properties: the function's context and the code that implements the function's behavior

Function objects

Note that each function object has a `prototype` property:

- This is not the same as the hidden prototype link to `Function.prototype`
- The `prototype` property links to an object
- This object has a `constructor` property whose value is the function

More about this when we discuss inheritance...

Function objects

As objects, functions can be used like any other value. A function can:

- be stored in a variable, object, or array
- be passed as an argument to a function
- be returned from a function
- have methods

Functions are special because they can be invoked.

Function literal

Function objects are created with function literals:

```
// Create a variable called add and store a function
// in it that adds two numbers
var add = function (a, b) {
    return a + b;
};
```

There are four parts to a function literal:

- The first part is the the reserved word `function`
- The second (optional) part is the name, used for recursion or by debuggers and other tools. A nameless function is **anonymous**.
- The third part is the set of zero or more parameters, wrapped in parentheses and separated by commas. They become local variables in the function, initialized by the caller.
- The fourth part is the body of the function, statements wrapped in curly braces that are executed when the function is invoked.

Function literal

A function literal can appear anywhere that an expression can appear. Functions can be defined inside of other functions:

- An inner function has access to its own parameters and variables
- An inner function also has access to the parameters and variables of the function it is nested within
- The function object created by a function literal has a hidden link to the context in which it was defined
- This is called a **closure**, and is the source of enormous expressive power

Invocation

Invoking a function suspends the execution of the current function and passes control to the new function. Each function receives two parameters in addition to those declared in the function definition:

- `this`: the `this` parameter is very important to object-oriented programming. There are four invocation patterns:
  - the method invocation pattern
  - the function invocation pattern
  - the constructor invocation pattern
  - the apply invocation pattern

  The value of `this` is determined by which of these invocation patterns was used to call the function.
- `arguments`: this bonus parameter is a list of the arguments. We will discuss it after the invocation patterns.

Invocation

The **invocation operator** is a pair of parentheses following any expression that produces a function value: a variable name that contains a function, a function literal, a function call that returns a function, . . .

The parentheses can contain zero or more arguments. Each of the arguments will be assigned to the function's parameter names. If they do not line up there is no runtime error:

- if there are too many arguments, the extra argument values will be ignored
- if there are too few arguments, the `undefined` value will be substituted for the missing values

There is no type checking on argument values. Any type of value can be passed to any parameter.

The method invocation pattern

When a function is stored as a property of an object, we call it a **method** of that object. When a method is invoked, `this` is bound to that object. If an invocation expression contains a refinement (a . expression or [*subscript*] expression), it is invoked as a method:

```
var myObject = {
    value: 0;
    increment: function (inc) {
        this.value += typeof inc === 'number' ? inc : 1;
    }
};

myObject.increment();
document.writeln(myObject.value);        // 1

myObject.increment(2);
document.writeln(myObject.value);        // 3
```

This binding of `this` happens at invocation time; this late binding makes these **public methods** highly reusable.

The function invocation pattern

When a function is not the property of an object, then it is invoked as a function:

```
var sum = add(3, 4);     // sum is 7
```

In this case, `this` is bound to the global object. This is a mistake. When an inner function is invoked, `this` should retain the `this` binding of the enclosing function:

```
myObject.double = function () {
    var helper = function () {
        this.value = add(this.value, this.value);
    };
    helper();            // invoke helper as a function
};
myObject.double();       // invoke double as a method
document.writeln(myObject.getValue());
```

This code is broken; `this` refers to the wrong thing inside helper.

The function invocation pattern

Fortunately, there is an easy workaround. Normal variables in the containing function are accessible to the inner function:

```
myObject.double = function () {
    var that = this;    // Workaround
    var helper = function () {
        that.value = add(that.value, that.value);
    };
    helper();           // invoke helper as a function
};
myObject.double();      // invoke double as a method
document.writeln(myObject.getValue());
```

This version works as expected.

The constructor invocation pattern

JavaScript is a **prototypal** inhertance language. This means that an object inherits properties directly from other objects. The language is class-free.

Most languages today are class-based. JavaScript uses some syntax borrowed from class-based languages to make programmers feel more comfortable, but this is just confusing.

If a function is invoked with the `new` prefix, then a new object will be created with a hidden link to the value of the function's `prototype` member, and `this` will be bound to that new object.

In addition, the `new` prefix changes the behavior of the `return` statement.

The constructor invocation pattern

```
// Create a constructor function call Quo.
// It makes an object with a status property.
var Quo = function (string) {
    this.status = string;
};

// Give all instances of Quo a public method called get_status.
Quo.prototype.get_status = function () {
    return this.status;
};

// Make an instance of Quo.
var myQuo = new Quo('confused');

document.writeln(myQuo.get_status());   // confused
```

Functions used with `new` are called **constructors**, and by convention are kept in a variable with a capitalized name. If called without the `new` prefix, bad things can happen without a warning or error.

Using this style of constructor functions is not recommended.

The apply invocation pattern

JavaScript allows functions to have methods.

The `apply` method lets us construct an array of arguments to use to invoke a function. It also lets us choose the value of `this`:

```
// Make an array of 2 numbers and add them.
var array = [3, 4];
var sum = add.apply(null, array);        // sum is 7

// Make an object with a status member
var statusObject = {
    status: 'A-OK'
};

// statusObject does not inherit from Quo.prototype, but we can
// invoke the get_status method on statusObject even though
// statusObject does not have a get_status method
var status = Que.prototype.get_status.apply(statusObject); // A-OK
```

The first argument to `apply` is bound to `this`, and the second is an array of parameters.

Arguments

`arguments` is a bonus parameter that all functions have available. It contains the complete list of arguments supplied when the function was invoked, including excess arguments that were not assigned to parameters:

```
// note: the variable sum inside the function does not interfere
// with the sum defined outside the function. The function only
// sees the inner one.
var sum = function () {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i += 1) {
        sum += arguments[i];
    }
    return sum;
};
document.writeln(sum(4, 8, 15, 16, 23, 42)); // 108
```

JavaScript design error: `arguments` is an array-like object, but not really an array. It has a `length` property, but it lacks all of the array methods.

Return

When a function is invoked, it begins execution with the first statement, and ends when it hits the `}` that closes the function body. Then control is returned to the part of the program that invoked the function.

The `return` statement causes the function to return early. When `return` is executed, the function returns immediately without executing the remaining statements.

A function always returns a value. If the `return` value is not specified, then `undefined` is returned.

If the function is invoked with the `new` prefix and the return value is not an object, then `this` (the new object) is returned instead.

Exceptions

JavaScript supports exceptions. An exception is an unusual mishap that interferes with the normal flow of a program. When such a mishap is detected, your program should throw an exception:

```
var add = function (a, b) {
    if (typeof a !== 'number' || typeof b !== 'number') {
        throw {
            name: 'TypeError',
            message: 'add needs numbers'
        };
    }
    return a + b;
};
```

The `throw` statement interrupts execution of the function. It should be given an exception object containing a `name` property that identifies the type of the exception and a descriptive `message` property. You can also add other properties.

Exceptions

The exception object will be delivered to the `catch` clause of a `try` statement:

```
// Make a try_it function that calls the new add
// function incorrectly
var try_it = function ( ) {
    try {
        add('seven');
    } catch (e) {
        document.writeln(e.name + ': ' + e.message);
    }
};

tryIt();
```

If an exception is thrown within a `try` block, control will go to its `catch` clause.

A `try` statement has a single `catch` block that will catch all exceptions. You can use the `name` property to identify different exception types.

Augmenting types

We can augment the basic types of JavaScript by adding methods to the right prototype objects. This works for:

- objects
- functions
- arrays
- strings
- numbers
- regular expressions
- booleans

In each case, modifying the global prototype object effectively modifies every instance of that type, including instances that already exist.

Augmenting functions

We can add a function to `Function.prototype`, making a new method available for all functions:

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

Now all functions have a method called `method` that lets us avoid typing the `prototype` property name.

Augmenting numbers

Getting just the integer part of a JavaScript number can be a bit clumsy, so we can add a new method to numbers:

```
Number.method('integer', function () {
    return Math[this < 0 ? 'ceiling' : 'floor'](this);
});

document.writeln((-10 / 3).integer());  // -3
```

This uses either `Math.ceiling` or `Math.floor` depending on whether the value is positive or negative.

Augmenting strings

JavaScript lacks a method that removes spaces from the ends of a string. This is an easy oversight to fix:

```
String.method('trim', function () {
    return this.replace(/^\s+|\s+$/g, '');
});

document.writeln('"' + "    neat    ".trim() + '"');
```

`trim` uses a regular expression, which we will discuss later.

Defensive augmentation

Augmenting basic types can add significantly to the expressiveness of the language. The prototype inheritance system means that all existing values instantly gain the new functionality.

Because the prototypes of basic types are public structures, we must be careful when mixing libraries. A defensive strategy is usually a good idea:

```
// Add a method if it does not already exist
Function.prototype.method = function (name, func) {
    if (!this.prototype[name]) {
        this.prototype[name] = func;
    }
};
```

Also, be careful to note that properties in the prototype chain show up in `for in` loops. Use the `hasOwnProperty` method to screen out inherited properties.

Recursion

A **recursive** function is a function that calls itself, either directly or indirectly. Recursion is a powerful programming technique in which a problem is divided into a set of similar subproblems, each solved with a simple solution. Generally, a recursive function calls itself to solve its subproblems.

The Towers of Hanoi is a famous puzzle with a simple recursive solution:

```
var hanoi = function hanoi(disc, src, aux, dst) {
    if (disc > 0) {
        hanoi(disc - 1, src, dst, aux);
        document.writeln('<pre>Move disc ' + disc +
                         ' from ' + src + ' to ' + dst + '</pre>');
        hanoi(disc - 1, aux, src, dst);
    }
}

hanoi(3, 'Src', 'Aux', 'Dst');
```

Towers of Hanoi

This produces the following solution for three discs:

```
Move disc 1 from Src to Dst
Move disc 2 from Src to Aux
Move disc 1 from Dst to Aux
Move disc 3 from Src to Dst
Move disc 1 from Aux to Src
Move disc 2 from Aux to Dst
Move disc 1 from Src to Dst
```

It works by breaking the problem into three subproblems:

- First, it uncovers the bottom disc by moving the substack above it to the auxiliary post.
- Next, it moves the bottom disc to the destination post.
- Finally, it moves the substack from the auxiliary post to the destination post.

Moving a substack is a job for a recursive call.

Recursive DOM functions

The browser's DOM is a tree, and recursion is a natural tool for manipulating trees:

```
var walk_the_DOM = function walk(node, func) {
    func(node);
    node = node.firstChild;
    while (node) {
        walk(node, func);
        node = node.nextSibling;
    }
};
```

This function visits every node of the tree in HTML source order, starting from some given node. It invokes a function, passing it each node in turn.

Recursive DOM functions

An example of a function that uses `walk_the_DOM` to gather a list of nodes containing a given attribute, optionally requiring a specific value for that attribute:

```
var getElementsByAttribute = function (att, value) {
    var results = [];

    walk_the_DOM(document.body, function (node) {
        var actual = node.nodeType === 1 && node.getAttribute(att);
        if (typeof actual === 'string' &&
                (actual === value || typeof value !== 'string')) {
            results.push(node);
        }
    });

    return results;
};
```

Note that we create the callback function as a function value, right when it is needed. This is a common style in JavaScript programming.

Tail recursion

Some languages offer **tail call optimization**. This means that if a function returns the result of invoking another function as its final result (including calling itself recursively),the invocation is replaced with a loop, preventing the stack from growing unnecessarily. Unfortunately, JavaScript does not currently offer this optimization:

```
var factorial = function factorial(i, a) {
    a = a || 1;
    if (i < 2) {
        return a;
    }
    return factorial(i - 1, a * i);
};

document.writeln(factorial(4)); // 24
```

The recursive call is the last thing to happen in the function, but JavaScript fails to notice it and apply an optimization. This means you must be wary of stack growth in similar situations.

Scope

Scope controls the visibility and lifetimes of variables and parameters:

```
var foo = function () {
    var a = 3, b = 5;

    var bar = function () {
        var b = 7, c = 11;          // a = 3,  b = 7, c = 11
        a += b + c;                 // a = 21, b = 7, c = 11
    };
                                    // a = 3,  b = 5, c is not defined
    bar();                          // a = 21, b = 5
};
```

JavaScript has function scope, but not block scope. Parameters and variables defined in a function are not visible outside the function, and a variable defined anywhere within a function is visible everywhere within the function.

To avoid confusion with block scope languages, it may be a good idea to declare all variables at the top of the function body.

Closures

One of the most important features of inner functions is that they have access to the parameters and variables of the functions they are defined within (with the exception of `this` and `arguments`).

`getElementsByAttribute` made use of this property to access the `results` variable. The more interesting case is when the inner function outlives the outer function:

```
var myObject = (function () {
    var value = 0;
    return {
        increment: function (inc) {
            value += typeof inc === 'number' ? inc : 1;
        },
        getValue: function () {
            return value;
        }
    }
})();
```

Closures

Private fields in C++ and Java can be simulated using this property:

```
// Create a maker function called quo. It makes an object
// with a get_status method and a private status property.
var quo = function (status) {
    return {
        get_status: function () {
            return status;
        }
    };
};

// Make an instance of quo
var myQuo = quo('amazed');

document.writeln(myQuo.get_status());
```

The `quo` function is effectively a constructor, but is not used with the `new` keyword, so it is not capitalized.

Closures

A function **captures** the environment in which it is defined. This is called a **lexical closure**, or just a closure.

```
var fade = function (node) {
    var level = 1;
    var step = function () {
        var hex = level.toString(16);
        node.style.backgroundColor = '#FFFF' + hex + hex;
        if (level < 15) {
            level += 1;
            setTimeout(step, 100);
        }
    };
    setTimeout(step, 100);
};

fade(document.body);
```

`setTimeout` calls the given function after the given number of milliseconds.

Bad example

The inner function has access to the actual variables of the outer functions, not copies:

```
var add_the_handlers = function (nodes) {
    var i;
    for(i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = function (e) {
            alert(i);
        }
    }
};
```

This function is assigns an event handler function to an array of nodes. When you click on a node, an alert box is supposed to display the ordinal of the node.

Instead, it always displays the total number of nodes. What went wrong?

Fixed example

This version fixes the problem:

```
var add_the_handlers = function (nodes) {
    var i;
    for(i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = function (i) {
            return function (e) {
                alert(i);
            };
        }(i);
    }
};
```

We create a new function each time and immediately invoke it, creating a new `i` variable that hides the old one from the handler function.

Be careful: this is an easy mistake to make.

Callbacks

Functions make it easier to deal with discontinous events. Suppose you start with a user interaction, contact the server, then display the server's response:

```
request = prepare_the_request();
response = send_request_synchronously(request);
display(response);
```

This approach will freeze the client while waiting for the network and the server. A better approach:

```
request = prepare_the_request();
send_request_asynchronously(request, function (response) {
    display(response);
});
```

By passing a function parameter and making the call asynchronously, the client can continue as normal while waiting for the response. The function will be invoked when the response is ready.

Modules

A module is a function or object that presents an interface by hides its state and implementation. We can use functions and closures to create them. This technique can eliminate many uses of global variables.

Consider a `deentityify` method for strings. Its job is remove HTML entities in a string and replace them with their equivalent characters, e.g., converting `'&quot;'` to `'"'`. An object mapping entity names to characters would be useful, but where to put it?

- keep it in a global variable: yuck
- define it in the function itself: runtime overhead because the object literal must be evaluated every time the function is called
- hidden in a closure: now we're talking

Modules

```
String.method('deentityify', function () {
    // The table of entity names and characters
    var entity = { 'quot': '"', lt: '<', gt: '>' };

    // Return the deentityify method
    return function () {

        // This is the deentityify method. It searches for
        // substrings that start with '&' and end with ';'. If the
        // characters in between are in the entity table, then
        // replace the entity with the character from the table. It
        // uses a regular expression (covered later).
        return this.replace(/&([^&;]+);/g,
            function (a, b) {
                var r = entity[b];
                return typeof r === 'string' ? r : a;
            }
        );
    };
}());
```

Note the last line. The first function returns the actual method.

Modules

We can now use this method:

```
document.writeln('&lt;&quot;&gt;'.deentityify());    // <">
```

The module pattern uses closures and scoping rules to hide the entity object. It is created only once, but only the method can access it.

The general pattern is:

- Create a function that
- defines private variables and functions;
- creates privileged functions, which, through closure will have access to the private variables and functions; and that
- returns the privileged functions or stores them in an accessible place.

Modules as secure objects

Say we want to generate serial numbers:

```
var serial_maker = function () {
    var prefix = '';
    var seq = 0;
    return {
        set_prefix: function (p) {
            prefix = String(p);
        },
        set_seq: function (s) {
            seq = s;
        },
        gensym: function () {
            var result = prefix + seq;
            seq += 1;
            return result;
        }
    };
}
```

Modules as secure objects

The serial numbers have a prefix and a unique number. Because the methods do not make use of `this`, there is no way to compromise the sequence, except through the defined interface, i.e., no fancy use of the `apply` invocation pattern:

```
var seqer = serial_maker();
seqer.set_prefix('Q');
seqer.set_seq(1000);
var unique = seqer.gensym();     // unique is "Q1000"
```

The methods themselves could be replaced, but that would still not permit access to `prefix` and `seq` except through the interface.

Cascade

Some methods do not have a return value. If we have them return `this` instead of `undefined`, we can cascade method calls:

```
getElement('myBoxDiv').
    move(350, 150).
    width(100).
    height(100).
    color('red').
    border('10px outset').
    padding('4px').
    appendText('Please stand by').
    on('mousedown', function (m) {
        this.startDrag(m, this.getNinth(m));
    }).
    on('mousemove', 'drag').
    on('mouseup', 'stopDrag').
    tip('This box is resizeable');
```

In this example, each of the methods returns the object, so we can immediately call another method in a chain. jQuery makes extensive use of this trick.

Curry

Functions are values. **Currying** allows us to produce a new function by combining a function and an argument:

```
var add1 = add.curry(1);
document.writeln(add1(6));        // 7
```

`add1` is a function that is the same as `add` only with one argument already provided. We can write the `curry` function to make this possible:

```
Function.method('curry', function ( ) {
    var args = arguments, that = this;
    return function ( ) {
        return that.apply(null, args.concat(arguments));
    };
});      // Something isn't right
```

Unfortunately, the `arguments` array is not actually an array, so it does not have the `concat` method.

Curry

To work around the lack of a `concat` method for the `argument` list, we apply the array `slice` method on both of the `argument` arrays. This produces real arrays that behave correctly with the `concat` method:

```
Function.method('curry', function ( ) {
    var slice = Array.prototype.slice,
        args = slice.apply(arguments),
        that = this;
    return function ( ) {
        return that.apply(null,
                args.concat(slice.apply(arguments)));
    };
});
```

Memoization

Functions can use objects to remember the results of previous operations, making it possible to avoid unnecessary work. This is called **memoization**:

```
var fibonacci = function (n) {
    return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);
};

for (var i = 0; i <= 10; i += 1) {
    document.writeln('// ' + i + ': ' + fibonacci(n));
}
```

This code works, but it does a log of unnecessary work. The `fibonacci` function is called 453 times: 11 times by us, 442 times in recursive calls, most of which are duplicating work.

Memoization

We can keep our memoized results in a `memo` array that we can hide in a closure. When our function is called, it first looks to see if it already knows the result. If so, it can return it immediately:

```
var fibonacci = function () {
    var memo = [0, 1];
    var fib = function (n) {
        var result = memo[n];
        if (typeof result !== 'number') {
            result = fib(n - 1) + fib(n - 2);
            memo[n] = result;
        }
        return result;
    };
    return fib;
}();
```

This version gives the same results, but `fibonacci` is only called 29 times: 11 times by us and 18 times to obtain the previously memoized results.

Memoization

We can generalize this:

```
var memoizer = function (memo, fundamental) {
    var shell = function (n) {
        var result = memo[n];
        if (typeof result !== 'number') {
            result = fundamental(shell, n);
            memo[n] = result;
        }
        return result;
    };
    return shell;
};
```

The initial `memo` array and the "real" function are passed is as parameters. The code manages the `memo` array and only calls the fundamental function when a result is absent from the array.

Memoization

We can define `fibonacci` with the memoizer by providing it with the initial `memo` array and the `fundamental` function:

```
var fibonacci = memoizer([0, 1], function (shell, n) {
    return shell(n - 1) + shell(n - 2);
});
```

By writing functions that produce other functions, we can reduce the amount of work we need to do. For example, to produce a memoizer factorial function, we only need to supply the basic factorial formula:

```
var factorial = memoizer([1, 1], function (shell, n) {
    return n * shell(n - 1);
});
```