Core JavaScript Documentation

Release 0.0

Jonathan Fine

Jun 28, 2017

Contents

This document has a search.

CHAPTER 1

About this course

Getting started

What you need

You'll need on your machine

1. A JavaScript interpreter which provides a command line
2. An editor on your machine.

You'll also to download and unzip the course `work folder`.

Windows and Linux

For Windows the easiest thing to do is to download the JSDB interpreter, and use notepad (or some other editor).

For Linux you and install Rhino and use your favourite editor.

```
$ sudo apt-get install rhino
```

Are you ready?

You'll want to be able to run the JavaScript interpreter from the command line when in the work folder. This will be automatic (via the PATH) with Linux and Rhino. For Windows the easiest thing to do is to place the jsdb.exe file in the work folder.

When you're ready type *js* at a command prompt. This starts the interpreter and gives you a *js>* prompt. You'll get something like this.

```
core-javascript-work$ js
Rhino 1.7 release 2 2010 01 20
js>
```

Now type the command as below at the JavaScript prompt (with Return at the end of each line) and you'll get responses as below.

```
js> a = '0'
0
js> b = 0
0
js> c = ''

js> a == b
true
js> b == c
true
js> a == c
false
```

You might be surprised by the last response from the interpreter, but every JavaScript interpreter does this.

Yes, you're ready

To exit the interpreter use Ctrl-C or Ctrl-D.

Counters example

Goal

The goal is to create a web page which contains several independent counters. Each time a counter is clicked, it is incremented. Here's you can try out a working example of what's wanted.

Below is the complete code of this example. To simplify the matter, it is completely self-contained. It uses no library code, other than the definition of *SimpleClass*.

In general library code is a good idea. This example is designed to teach you the basics of JavaScript, and not the use of a library. We hope that what you learn here will help you choose a library, and build libraries of your own.

counters.html

```
<html>
<head>
<script src="library.js"></script>
<script src="counters.js"></script>
<link rel="stylesheet" type="text/css" href="counters.css" />
<title>JS for Python: example: counters</title>
</head>
<body>
<h1>Counters</h1>

<p>Click on a counter to increment its value.</p>

<div id="example">
<p>This will disappear if JavaScript is working properly.</p>
</div>
```

```
<p>Return to documentation of <a href="../counters-example.html">Counters example</a>.
↪</p>
</body>
```

counters.css

```
body {
    background: #DDD;
    font-family: sans-serif;
}

#example {
    padding: 20px;
}

#example span {
    padding: 10px;
    margin: 10px;
    border: 10px solid blue;
    background: #DDF;
    foreground: blue;
    font-weight: bold;
}
```

counters.js

```
(function()
{
    // Define a Counter class.
    var counter = {};           // Prototype object for Counter.

    counter.__init__ = function(name){

        this.name = name;
        this.count = 0;
    };

    counter.onclick = function(event){

        this.count ++;
    };

    counter.html = function(){

        return this.name + ' ' + this.count;
    };

    Counter = SimpleClass(counter);


    // Make explicit use of global variables.
    var global = (function(){return this;})();
```

```
    // Interaction.
    var onclick_factory = function(models){

        var onclick = function(event){

            event = event || global.event; // For IE event handling.
            var target = event.target || event.srcElement;
            var id = target.id;
            if (id) {
                var id_num = +id.slice(1);
                var model = models[id_num];
                model.onclick();
                var html = model.html();
                if (html){
                    global.document.getElementById(id).innerHTML = html;
                }
            }
        };
        return onclick;
    };


    // Set up the web page.
    global.onload = function(){

        var models = [
            Counter('apple'),
            Counter('banana'),
            Counter('cherry'),
            Counter('date')
        ];

        var element = document.getElementById('example');

        element.innerHTML = (
            '<span id="a0">apple 0</span>'
            + '<span id="a1">banana 0</span>'
            + '<span id="a2">cherry 0</span>'
            + '<span id="a3">date 0</span>'
        );

        element.onclick = onclick_factory(models);
        element = undefined;    // Avoid IE memory leak.
    };

})();
```

Test tools

To make it easier to test code, and to ensure that example code is correct, there is a small testtools.js file in the work folder.

How to write tests

To prepare a test create a file like this. It's also in the work folder. (For real examples the file will give *assert* a more interesting argument, whose truth or falsity is perhaps not obvious.)

```
TEST('all-pass', function()
{
    assert( 2 + 2 === 4 );
    assert( 'the ' + 'cat' === 'the cat');
});


TEST('2-4-fail', function()
{
    assert( true );
    assert( false );
    assert( 1 );
    assert( 0 );
});
```

How to run tests

Here's how to run the tests at the command line, finishing with a command prompt that allows you inspect the state and also run the test again.

Rhino

For Rhino use *load* to run the test again.

```
core-javascript-work$ js -f testtools.js -f demo_testtools.js -f -
Testing: all-pass
Testing: 2-4-fail
! assert 2 has failed
! assert 4 has failed
Rhino 1.7 release 2 2010 01 20
js> load('demo_testtools.js')  // You type this.
Testing: all-pass
Testing: 2-4-fail
! assert 2 has failed
! assert 4 has failed
js>
```

JSDB

For JSDB use *run* to run the test again.

```
C:core-javascript-work> jsdb -load testtools.js -load demo_testtools.js
Testing: all-pass
Testing: 2-4-fail
! assert 2 has failed
! assert 4 has failed
js> load('demo_testtools.js')  // You type this.
Testing: all-pass
Testing: 2-4-fail
```

```
! assert 2 has failed
! assert 4 has failed
js>
```

testools.js

(Optional on a first reading.) Here's the file testools.js. It's short and simple (and has a not-nice dependency on global variables). It also has a branch based on whether it's being run on JSDB or Rhino.

```
var telltail = 'global';
var _assertion_count = 0;

// Define a global function 'log'.
var log;
if (this.hasOwnProperty('jsArguments')){

    log = function(s){
        print(s + '\n');
    };
} else {
    log = print;
}

//
var TEST = function(title, code){

    var x;

    log( 'Testing: ' + title );
    _assertion_count = 0;

    try {
        code();
    } catch (x) {
        log('! Exception at ' + x.fileName +':' + x.lineNumber);
        log('! [' + x.name + '] '  + x.message );
    }

};

var assert = function(arg){
    _assertion_count += 1;
    if (!arg){
        var msg = '! assert ' + _assertion_count + ' has failed';
        log(msg);
    }
};
```

CHAPTER 2

Objects

Immutables

First, a quick word about variables. A value is an indentifier that holds a value. In JavaScript every value has a type, but variables are not typed. In other words, any variable can hold any value. Most dynamic language (such as Perl, Python and Ruby) are like this.

Always declare variables with a var statement, because sometimes a missing var can cause a hard-to-find bug. Experiments at the command line are the only exception to this rule.

We can change an array or a 'dictionary'. Immutable values are values that can't be changed. We can't change, for example, the third character in a string.

Strings

Strings literals are delimited by single or double quote marks, with special characters escaped. There's no difference between the two forms (except in single quote you don't have to escape double quote, and vice versa.) Generally, I prefer the single quote form as it's less busy on the eye and slightly easier to type.

```
js> s = 'Hello world.'
Hello world.
```

We can add two strings together to produce a third string.

```
js> r = "I'm me. "
I'm me.
js> r + s
I'm me. Hello world.
```

typeof

There's a built-in operator called 'typeof' that returns a string that, sort-of, gives the type of a value.

```
js> typeof(s)
string
```

Because typeof is an operator (just as '+' is an operator) the parentheses are not needed, and many JavaScript programmers omit them.

```
js> typeof s
string
```

Here we see that typeof produces (returns) a string.

```
js> typeof typeof s
string
```

Numbers

JavaScript numbers are platform and processor independent. It uses IEE 754 to represent both integers and floats.

```
js> i = 42
42
js> typeof i
number
```

Booleans

JavaScript has keywords 'true' and 'false' whose values are always true and false respectively.

```
js> t = true
true
js> typeof b
boolean
```

Logical comparisions also produce Booleans.

```
js> f = (1 > 2)
false
js> typeof f
boolean
```

undefined and null

JavaScript has *two* values that represent None. Later, we'll see why, and which to use when. For now we'll simply note that they are different, because their types are different.

```
js> typeof undefined
undefined
js> typeof null
object
```

Gotchas

```
js> s = 'Hello world.'
Hello world.
js> s.lang = 'en'
en
js> s.lang === undefined
true
```

Equality

Equality in JavaScript can be a little odd.

Double equals

JavScript has two operations for equality (and inequality). One of them is '==' (and '!=' for inequality). It's the oddest.

Not transitive

If *a* is equal to *b*, and if *b* is equal to *c* , then we expect *a* to be equal to *c*. This is called the transitive property.

```
js> '0' == 0       // 'a' is equal to 'b'
true
js> 0 == ''       //  'b' is equal to 'c'
true
js> '0' == ''     // 'a' is not equal to 'c'
false
```

Not reflexive

We expect *a* to be equal to *a*. This is called the reflexive property.

```
js> NaN == NaN
false
```

Fortunately, this seems to be the only example.

Is symmetric

If *a* is equal to *b* then we expect *b* to be equal to *a*. This is called the symmetric property. In JavaScript equality is symmetric.

Triple equality

JavaScript also has '===' (and '!==' for inequality).

Is transitive

Unlike double equals, triple equals is always transitive.

```
js> '0' === 0       // unequal
false
js> 0 === ''        // unequal
false
js> '0' === ''      // unequal
false
```

Not reflexive

```
js> NaN === NaN
false
```

By the way, there's a thread on Facebook with subject Time and Date on my wall shows NaNNaNNaN at NaN:NaN. I wonder how that happened.

Is symmetric

Triple equality is still symmetric.

Don't use double equality

Double equality does implicit conversions and besides has some odd rules. My advice is **don't use double equality**.

Triple equality does not do conversions. If you want to do conversions in your comparision my advice is to make them explicit.

Compare as string

The easiest conversion is to string. Here are two immutables, a number and a string.

```
js> a = 0
0
js> b = '0'
0
```

These quantities are double equal but not triple equal.

```
js> a == b
true
js> a === b
false
```

Here's how to do an explicit conversion to string before comparison, which gives an equality.

```
js> '' + a === '' + b     // Converts variables to strings.
true
```

Note: It's always easier to read a triple equal comparison, because you're not distracted by the complex double equal rules.

Compare as number

[To follow later.]

Objects

JavaScript has objects and is object-oriented, but in an unusual way (more on that later).

Simple objects

Simple objects are like dictionaries or hashes in other languages. They support key-value storage and access to attributes. Here's how to create a simple object.

```
js> obj = {}
[object Object]

js> typeof obj
object
```

We can store attribute values in a simple object (or any object in the doc:*object-tree*).

```
js> obj.s = 'hi'
hi
js> obj.i = 10
10
```

We can get these values back again.

```
js> obj.s === 'hi'
true
js> obj.i == 10
true
```

Missing attributes

It's not an error to ask for something that's not there. We get undefined.

```
js> obj.dne === undefined
true
```

We still get undefined if we set the value to undefined.

```
js> obj.undef = undefined
undefined
js> obj.dne === undefined
true
```

You can use null to signal that there is a value, but that it is None.

```
js> obj.none = null
null
js> obj.none === null
true
```

hasOwnProperty

You can use hasOwnProperty to help figure out why an object has an undefined attribute. It's also useful when inspecting the doc::*object-tree*. Often, however, it's better to write your code so you don't need to do this (for example by using null).

```
js> obj.hasOwnProperty('dne')
false
js> obj.hasOwnProperty('undef')
true
js> obj.hasOwnProperty('none')
true
```

Object literals

You can create a simple object by placing key-value pairs in the curly braces.

```
js> someone = {
  >     'name': 'Joe Doe',
  >     'age': 43
  > }
[object Object]

js> someone.age
43
```

Take care *not* to put a trailing semicolon in the object literal. It will work in Firefox but not in Internet Explorer.

Arrays

An array is a list of items. You can put anything is as a value for the list. Use square brackets to create an array.

```
js> array = []
```

Arrays expand to accomodate the data you store in them. You can even leave gaps.

```
js> array[0] = 'zero'
zero
js> array[3] = 'three'
three
```

An array turned into a string consists of the string on its entries joined by commas.

```
js> array
zero,,,three
js> array[1] === undefined
true
```

Array literals

As with simple objects, simply place the values between the square brackets, separated by commas.

```
js> seasons = ['spring', 'summer', 'autumn', 'winter']
spring,summer,autumn,winter
```

As with simple objects, beware of trailing commas and missing entries. This will work in some browsers and not others.

JSON

JSON,stands for JavaScript Simple Object Notation. It is very much one of the best parts of JavaScript. It is the fat-free alternative to XML, and is widely used in AJAX (instead of XML). JSON objects are object and array literals constructed using only simple objects, arrays, strings, true, false and null.

If your code accepts JSON objects then it will be a lot easier to use it with AJAX or otherwise integrate it with other systems.

Many programming languages have JSON libraries. You don't have to use JavaScript to use JSON.

To provide standards, there are rules on how to write a JSON objects. Many applications and programming languages will generate valid JSON for you. I find YAML a convenient way of authoring JSON data.

The object tree

Objects can have attributes. The statements

```
value = obj.attr              # Get the attr of 'a'.
obj.attr = value               # Set the attr of 'a'.
```

respectively **set** and **get** the *attr* attribute of the object *obj*. Inheritance is where an object gets some its attributes from one or other more objects. JavaScript uses an object tree for inheritance.

Tree

All JavaScript objects are part of an inheritance tree. Each object in the tree has a parent object, which is also called the prototype object (of the child). There is a single exception to this rule, which is the **root** of the tree. The root of the tree does not have a parent object.

Note: You can't get far in JavaScript without understanding the object tree.

Get

When JavaScript needs to get an attribute value of an object, it first looks up the attr of the attribute in the object's dictionary. If the attr is a key in the dictionary, the associated value is returned.

If the attr is not a key, then the process is repeated using the object's parent, grandparent, and so on until the key is found. If the key is not found in this way then *undefined* is returned.

Set

When JavaScript needs to set an attribute value of an object it ignores the inheritance tree. It simply sets that value in the object's dictionary.

Root

When the interpreter starts up, the **root** of the tree is placed at *Object.prototype*. (We'll find out later why that location is used.)

Every object inherits from the **root**, although perhaps not directly. Here's a simple example:

```
js> root = Object.prototype
js> a = {}
js> a.attr === undefined
true
js> root.attr = 'gotcha'
js> a.attr
gotcha
```

If we give *root* a *attr* attribute then every other object, including those already created and those not yet created, also has a *attr* attribute with the same value. (In practice it's better not to change Object.prototpye.)

This applies arrays:

```
js> array = [0, 1, 2]
0,1,2
js> array.attr
gotcha
```

And even to functions:

```
js> f = function(){}
function () {
}

js> f.attr
gotcha
```

Note: A page might have many scripts, all of which would like to modify the Object.prototype root object. This can cause bugs and incompatibilities. So try not to do this.

However, this up-the-tree lookup does not apply if *attr* is found earlier in the tree. We continue the previous example to show this, and the behaviour of set.

```
js> a.attr = 'fixed'
js> a.attr
fixed
js> root.attr
gotcha
```

Create

Any tree can be constructed from its root, together with a command **create(parent)** that returns a new **child** of the given parent node.

In all but the most recent version of JavaScript the *create* function is not built in. However, it's easy to write one, once you know enough JavaScript. Here's how its done in the work folder's library.js file.

```
var create = function(parent){

    var tmp = function(){};
    tmp.prototype = parent;
    var child = new tmp();

    return child;
};
```

Using create

Here's an example of its use:

```
js> a = {}
js> b = create(a)
js> a.attr = 'apple'
apple
js> b.attr
apple
```

And a continuation of the example:

```
js> c = create(b)
js> c.attr
apple
js> b.attr = 'banana'
banana
js> c.attr
banana
```

Note: JavaScript uses an inheritance tree. By using *create*, we can create any inheritance tree. All JavaScript objects are in this tree.

Simple classes

JavaScript has objects, which through the object tree support inheritance. Almost all object-oriented languages have classes and instances, but JavaScript has neither. This will be explained later, along with a discussion of the consequences.

For now our focus is on getting going with object-oriented programming. The easiest way to do this is via create.

Definition

Here's a factory function that creates classes. It returns a function called *cls*, which in turn when called returns instances. At it's heart is *create*, and also *apply* and *arguments*, which we've not seen before).

```
var SimpleClass = function(prototype){

    var cls = function(){

        var instance = create(prototype);
        instance.__init__.apply(instance, arguments);
        return instance;
    };

    return cls;
};
```

Why *cls*?

We call the returned function *cls* because in JavaScript *class*, like *function* is a reserved word in JavaScript. However, in JavaScript the identifier *class* has no meaning, and cannot be used in any valid JavaScript program. It seems that they intended to provide a built-in class capability, but never got round to it.

Fruit example

Here's an example of the use of *SimpleClass*. There's a bug in it, and I've changed the test so it passes. Can you find the bug and fix the example?

```
TEST('SimpleClass', function()
{
    var fruit = {};

    fruit.__init__ = function(name, colour){

        this.name = name;
        this.colour = colour;
    };

    fruit.greet = function(){

        return "Hello, I'm a " + this.color + " " + this.name + ".";
    };

    var Fruit = SimpleClass(fruit);

    var bn = Fruit('banana', 'yellow');
    var rc = Fruit('cherry', 'red');
    var gs = Fruit('Granny Smith', 'green');

    assert( bn.greet() === "Hello, I'm a undefined banana." );

});
```

CHAPTER 3

Functions

The global object

JavaScript has a global object. It is, so to speak, the object of last resort. If there's no other suitable object then JavaScript will use the global object (rather than reporting an error).

Note: Douglas Crockford writes **JavaScript's global object ... is far and away the worst part of JavaScript's many bad parts.**

Getting global

Here's how to get the global object.

```
js> return_this = function(){return this;}

function () {
    return this;
}

js> global = return_this()
[object global]
```

The programmer who wants to can always obtain access to the global object.

Global variables

Global variables are nothing more than attributes of the global object.

```
js> s = 'Hello world.'
Hello world.
js> global.s
Hello world.
js> global.i = 42
42
js> i
42
```

Global pollution

It's very easy to inadvertently pollute (change) the global object. All it takes is an assignment to an undeclared variable in a function.

```
js> pollute = function(n){ i = n; };
function (n) {
    i = n;
}
js> i
42
js> pollute(13)
js> i
13
```

More global pollution

There's a more subtle way to pollute the global object, which involves *JavaScript's this object*.

To begin with, note that we can push values onto an array.

```
js> array = []

js> array.push(1, 2, 3)
3
js> array
1,2,3
```

Let's try using *array.push* as a stand-alone function.

```
js> p = array.push
function push() { [native code for Array.push, arity=1] }

js> p(4, 5, 6)
3
```

The push function returns the length of the object it has just pushed to. So the value *3* above is a signal that all is not well. And indeed it is not. The original array is unchanged, and the global object has an entry at 0.

```
js> array
1,2,3
js>
js> global[0]
4
```

It's also got a length!

```
js> global.length
3
```

Explanation

This behaviour is a consequence of *What is this?* and *Bind is transient*, together with JavaScript's no-exceptions design.

Closures

You'll see closures a lot in JavaScript. The reasons for this are:

1. In JavaScript, *Bind is transient*.
2. It is callback functions attached to DOM elements that respond to events.
3. Most code attaches callbacks to individual DOM elements (rather than delegating to a parent element).

Test code

Here's the test code for closures.

```
TEST('closures', function()
{
    // A factory for creating get-set pairs.
    var get_set_factory = function(){

        var value;
        var get = function(){
            return value;
        };
        var set = function(new_value){
            value = new_value;
        };

        return {
            get: get,
            set: set
        };
    };

    // Create and unpack two get-set pairs.
    var tmp;
    tmp= get_set_factory();
    var get_a = tmp.get;
    var set_a = tmp.set;

    tmp= get_set_factory();
    var get_b = tmp.get;
    var set_b = tmp.set;

    // Test that both pairs work.
```

```
    set_a(12);
    assert( get_a() === 12 );

    set_b(13);
    assert( get_b() === 13 );

    // Test that each pair has its own value.
    assert( get_a() === 12 );

});
```

Discussion

The factory function creates two functions, *get* and *set*, which refer to the variable *value* of the factory function. The *get* function returns the value of this variable, while the *set* function changes it.

To test the factory function we create two get-set pairs. We then test that each *set* operation changes the value return by its partner *get* operation.

Finally, we test that the two pairs don't interfere with each other.

Explanation

Objects continue to exist so long as a reference remains to them. When the last reference to an object is removed (or when the object is part of *cyclic garbage*) the object can be destroyed. When the return value of a function is stored (say as the value of a variable) then this return value continues to exist. Similarly, anything referred to by the return value continues to exist.

Closures come about when the return value of a function is itself a function defined within the outer function (or contains references to such functions).

Suppose the outer function returns an *inner function* (or in other words a function that is defined within the outer function). Suppose also that the inner function uses a variable (or parameter) of the outer function. In this situation the inner function holds a reference to the value of this variable. The code in the inner function can read (or get) this value. It can also write to this variable.

One final point. Each call of the outer function creates a new instantiation of the function variable.

Exercises

A programmer, perhaps in a hurry, misses out a *var* in the example above. So now it reads:

```
var get_set_factory = function(){

   value;   // Here's the missing *var*.
```

1. Which tests to you expect to pass, and which to fail?
2. Make this change and run the tests. Which actually fail?
3. Explain what is happening.
4. What does this tell us about coding and testing?

What is this?

JavaScript has a keyword *this*, whose value depends on the execution context. It is always an object (and never an immutable, such as a number).

Test code

```
TEST('return_this', function()
{
    var return_this = function(){
        return this;
    };

    var default_this = return_this();
    assert( default_this.telltail === 'global' );

    var obj = {};
    obj.return_this = return_this;
    assert( obj.return_this === return_this );

    assert( obj.return_this() === obj );
    assert( obj['return_this']() === obj );

    assert( return_this.call(obj) === obj );
    assert( return_this.apply(obj) === obj );

    assert( return_this.call(undefined).telltail === 'global' );
    assert( return_this.apply(undefined).telltail === 'global' );

});
```

Discussion

The function *return_this* is a convenient way of getting of getting hold of the object that is the value of *this*. The code that follows shows:

1. The default value of *this* is the global object.
2. When *return_this* is called as either a method (i.e. an attribute that is a function) or item of *obj* then *this* is *obj* itself.
3. That the *call* and *apply* methods of a function allow us to set the value of *this* within a particular execution of the function.
4. If we pass *undefined* to function *call* or *apply* then we get the global object as *this*.

Explanation

Throughout the execution of JavaScript code, the identifier *this* has a value, which is an object (and not an immutable value). The value of *this* depends on how the JavaScript interpreter got to be executing the code.

1. At the command line *this* is the global object.
2. In a function *this* is the object from which the interpreter got the function.

3. Because global variables are attributes of the global object, the rule for calling global variables as functions is a special case of the previous rule.
4.

Here's a helpful way to look at the situation. It is as if the interpreter maintains a *this stack*, which starts containing a single item, the global object. Each time an item access of an object is immediately followed by a function call, the object is added to the *this stack*, and removed at the end of the function call. For all other function calls the global object is added to the *this stack*.

During execution of a function the value of *this* is the element at the top of the *this stack*.

Finally, the function *call* and *apply* methods allow for a particular object to be placed at the top of the *this stack* before the function is called.

Exercises

1. At the command-line try making an assignment to *this*. What happens?
2. Try loading a JavaScript program that makes an assignment to *this*. What happens?
3. For what values of *value* is the following true?

```
return_this.apply(value) === value;
```

4. Write a simple test that detects these values.

Bind is transient

Test code

Here's the test code for attribute bind.

```
TEST('transient-bind', function()
{
    // Set up the test object.
    var return_this = function(){
        return this;
    };
    var obj = {
        return_this : return_this
    };
    assert( obj.return_this === return_this );

    // Deferring method call changes the outcome.
    var method_deferred = obj.return_this;
    assert( method_deferred ===  obj.return_this );
    assert( method_deferred() !==  obj.return_this() );  // AAA

    // The two outcomes, precisely stated.
    assert( obj.return_this() === obj );
    assert( method_deferred().telltail === 'global' );

    // Another view on the matter.
    assert( method_deferred === return_this );
    assert( method_deferred() === return_this() );  // BBB
```

```
});
```

Discussion

We create an object that has a *return_this* method. The outcome of calling *return_this* depends on how it is called.

There's a logic in the test code that produces contradictory, or at least surprising, outcomes.

From

we would expect

But this means that in *obj.return_this()* the *obj* is not relevant. But JavaScript (or its designers) wanted objects to have methods, so they introduced a special rule.

Explanation

JavaScript does not distinguish classes and instances. Python, and perhaps other dynamic languages, does. This allows Python to supply a bound method, which retains a reference to the object, when a function belonging to the class is called on an instance.

Here's a Python (version 3) command line dialogue that illustrates this.

```
>>> def method(): pass           # The function.
...
>>> class A: method = method     # The class.
>>> A.method                     # Function from class.
<function method at 0x15be518>
>>> A.method is method           # Same as the original function.
True
>>> a = A()                      # Instance.
>>> a.method                     # Bound method, not original function.
<bound method A.method of <__main__.A object at 0x16b9d10>>
```

The odd behaviour shown in the JavaScript test code is a consequence of JavaScript not distinguishing between classes and instances.

What is new?

This can be omitted at a first reading, and omitted altogether if you never have to deal with code that uses the *new* operator. If you do, then you **must** read about the missing *new* problem.

JavaScript has an object tree. This is a fundamental language feature. The *new* operator is a way of adding objects to the tree. The *create* function, which is built-in to the latest version of JavaScript, can be defined in terms of *new*, and vice versa.

Test code

```
TEST('new-operator', function()
{
    // A 'traditional class'.  It records 'this' as a side effect.
    var this_in_Fn;
```

```
    var Fn = function(){
        this_in_Fn = this;
    };

    // Call the 'class' without the 'new'.
    this_in_Fn = null;          // Clear any previous value.
    var non_instance = Fn();
    assert( non_instance === undefined );
    assert( this_in_Fn.telltail === 'global');

    // Call the 'class' with the 'new'.
    this_in_Fn = null;          // Clear any previous value.
    var instance = new Fn();
    assert( instance === this_in_Fn );
    assert( instance.telltail === undefined );

    // Demonstrate that instance is a descendant of Fn.prototype.
    Fn.prototype.telltail = 'instance-of-Fn';
    assert( instance.telltail === 'instance-of-Fn' );

    // We can give Fn a new prototype.
    instance.new_telltail = 'child-of-instance';
    Fn.prototype = instance;

    // And now an instance of Fn has two telltails.
    var second_instance = new Fn();
    assert( second_instance.telltail === 'instance-of-Fn' );
    assert( second_instance.new_telltail === 'child-of-instance' );

});
```

Discussion

By changing the prototype object of *Fn* we start building a tree of objects.

There is a built-in relation between an object and its parent. There is no relation between the object and its constructor, here called *Fn*, except that at the time of construction the parent of the object is Fn.prototype.

The exact rules for the behaviour of *new* are complicated and not given here.

No missing *new* warning

We saw that when *Fn* is called without the *new* then during its execution the value of *this* is the global object. If the constructor *Fn* mutates *this* (and almost every constructor does), then it is the global object that is mutated.

This is very bad:

1. You may clobber someone elses data.
2. Someone else may clobber your data.
3. If you create two instances, you will clobber your own data.
4. Testing for this error involves extra work.
5. Bugs that arise from this error can appear random and can be hard to find.

Note: Crockford writes **A much better alternative is to not use new at all."**

Arguments

This can be omitted at a first reading. The main point here is that in JavaScript

1. There is no checking for number and type of arguments.
2. The arguments actually supplied are stored in a special variable, called *arguments*.
3. The value of *arguments* is almost, but not exactly, an array.

Test code

```
TEST('return_arguments', function()
{
    var return_arguments = function(){
        return arguments;
    };

    var my_arguments = return_arguments(0, 1);

    assert( my_arguments.length === 2 );

    assert( my_arguments[0]  === 0 );
    assert( my_arguments[1]  === 1 );

    assert( '' + my_arguments === '[object Object]' );
    assert( '' + [0, 1] === '0,1' );

    var array_slice = [].slice;
    var my_arguments_fixed = array_slice.call( my_arguments );

    assert( '' + my_arguments_fixed === '0,1');

    var obj = {};
    assert( obj.toString.call( my_arguments_fixed ) === '[object Array]' );

});
```

Mark Miller's device

This can be omitted at a first reading. Mark Miller is one of the leading JavaScript experts at Google. He's been credited an ingenious application of call that gives a simple and reliable test for whether or not an object is an array.

This may not sound very much, but it is something that had been puzzling the other experts for some years. For example, Crockford in his *Good Parts* (published 2008, page 61) gives a much more complex and less reliable solution to this problem.

The moral of this story is that even the experts in JavaScript can have difficulty finding the best way to solve a simple problem.

Test code

```
TEST('millers-device', function()
{
    var array = [];
    var obj = {};
    var fn = function(){};

    assert( typeof array === 'object' );
    assert( '' + obj === '[object Object]' );

    assert( obj.toString() === '[object Object]' );

    var object_toString = obj.toString;

    assert( object_toString.call(array) === '[object Array]' );
    assert( object_toString.call(fn) === '[object Function]' );

});
```

CHAPTER 4

Examples and exercises

Exercise one

Here's some JavaScript code I found somewhere. I've changed the names of everything to hide the origin of the code. But it's not something I've made up.

The exercise is

1. Understand and describe what it does.
2. Provide clearer code that does the same thing.
3. Suggest a better way of going about things.

```
var AAA = (function (BBB)
{
    BBB.fff = function(ccc)
    {
        return {
            ggg: function(ddd)
            {
                return BBB.hhh(ddd, ccc.iii());
            }
        };
    };
    return BBB;

}(AAA || {}));
```

Hints

1. The code creates and calls an anonymous function, a bit like this.

```
js> a = function(x){return x + 1}(2)
3
```

2. What happens if AAA is an object?
3. What happens if AAA is undefined?
4. For now, ignore the assignment to BBB.fff. What value does AAA get? (The answer depends the initial value of AAA.)

Solution one

Here's the code we're studying.

```
var AAA = (function (BBB)
{
    BBB.fff = function(ccc)
    {
        return {
            ggg: function(ddd)
            {
                return BBB.hhh(ddd, ccc.iii());
            }
        };
    };
    return BBB;

}(AAA || {}));
```

Verbal summary

We are making sure that there's an object AAA and giving it an attribute AAA.fff, which is a function. We won't say more about the function in this summary.

Hoisting once-only arguments

First, we'll hoist the argument to the anonymous function into its body. We can always do this, without changing the meaning, provided the function is called only once.

```
var AAA = function ()
{
    var BBB = AAA || {};
    BBB.fff = function(ccc)
    {
        return {
            ggg: function(ddd)
            {
                return BBB.hhh(ddd, ccc.iii());
            }
        };
    };
    return BBB;
```

```
}();
```

There, that's better. We no longer have to read to the end of the function to find out what BBB is (and in particular its relationship with AAA).

What is AAA?

Next, we'll hide (or ignore) the assigment to BBB.fff. This assignment sets an attribute on the object pointed to by the variable BBB. In other words, it mutates the object pointed to by BBB, but the variable BBB still points to the same object as it did before.

```
var AAA = function ()
{
    var BBB = AAA || {};
    return BBB;
}();
```

If AAA starts out as an object then the expression

```
AAA || {}
```

evaluates to AAA, while if AAA is undefined then it evaluates to the newly created object literal (the left-and-right curly braces). In either case, this value is assigned to the local variable BBB, which is then returned by the anonymous function and bound to AAA.

Put another way, if AAA starts off as undefined then it is bound to a new object, while if it starts off as an object then it finishes as the same object.

Therefore, our truncated version of the original exercise is equivalent to

```
var AAA = AAA || {};
```

This idiom is very common in JavaScript, and is (as here) a consequence of using objects as namespaces (which is better than using the global object) and defensiveness about the order in which files are loaded.

Here's a command line session that shows this equivalence. To begin with AAA is undefined and the assignment binds AAA to a new object. For the second assignment AAA is defined and the assignment binds AAA to the object is already bound to.

```
js> var AAA = AAA || {}
js> AAA
[object Object]
js> pointer = AAA
[object Object]
js> var AAA = AAA || {}
js> pointer === AAA
true
```

The assignment to BBB.fff

Reading the code, it looks as if BBB is local to the anonymous function. As a variable it is, but its value is not. The value of BBB is bound to the global object AAA at the end of the function call.

The following is almost equivalent to our original JavaScript code. Note that BBB.hhh has been changed to AAA.hhh.

```
var AAA = AAA || {};

AAA.fff = function(ccc)
{
    return {
        ggg: function(ddd)
        {
            return AAA.hhh(ddd, ccc.iii());
        }
    };
};
```

Caveat

Why did I say *almost equivalent*? Because we might subsequently do something like

```
var CCC = AAA;
AAA = undefined;
```

For the original code the closure variable BBB still exists and points to the same object as CCC, even though AAA is undefined. Thus, the reference to BBB.hhh in the original code will continue to work.

For the revised code, when the value of AAA is set to undefined the reference to AAA.hhh will fail.

Exercise two

Here's some more JavaScript code I found. I've changed the names of everything to hide the origin of the code. But it's not something I've made up.

```
var copy_attributes = function(tgt, src){

    tgt.aaa = src.get_aaa();
    tgt.bbb = src.get_bbb();
    tgt.ccc = src.get_ccc();
    tgt.ddd = src.get_ddd();
    tgt.eee = src.get_eee();
    tgt.fff = src.get_fff();
    tgt.ggg = src.get_ggg();
    tgt.hhh = src.get_hhh();
};
```

The exercise is

1. Rewrite the function copy_attributes so that it makes a loop over

   ```
   var keys = ['aaa', 'bbb', 'ccc', 'ddd', 'eee', 'fff', 'ggg', 'hhh'];
   ```

2. Write a copy_attributes_factory function so that we can write

   ```
   var copy_attributes = copy_attributes_factory(keys);
   ```

3. Write a Fields class such that something like this will work:

```
var aaa = Fields(keys);
aaa.copy_attributes(src, tgt);
```

Hints

1. In JavaScript attribute access and item access are the same. First we use item access to set:

```
js> a = {}
[object Object]
js> key = 'attrname'
attrname
js> a[key] = 42
42
```

   and then we use attribute access to get:

```
js> a.attrname
42
```

2. This applies even for function (aka method) calls.
3. Have the factory function store the keys in a closure.
4. There are many ways of writing Fields. Choose one that suits you.

Solution two

Here's the code we are studying. The first task is make a loop that does the assignments.

```
var copy_attributes = function(tgt, src){

    tgt.aaa = src.get_aaa();
    tgt.bbb = src.get_bbb();
    tgt.ccc = src.get_ccc();
    tgt.ddd = src.get_ddd();
    tgt.eee = src.get_eee();
    tgt.fff = src.get_fff();
    tgt.ggg = src.get_ggg();
    tgt.hhh = src.get_hhh();
};
```

Making a loop

As previously hinted, we use item access rather than attribute access. The problem with attribute access is that the name of the key is hard-coded and we want it to vary during the loop.

Here's the solution. There's a bit of overhead in setting up the loop, but the body of the loop is straightforward.

```
var copy_attributes = function(tgt, src){

    var keys = ['aaa', 'bbb', 'ccc', 'ddd', 'eee', 'fff', 'ggg', 'hhh'];
    var i;
    var key;
```

```
    for(i=0; i< keys.length; i++){
        tgt[key] = src['get_' + key]();
    }
};
```

Factory function

The second task is to write a factory function that takes the list of keys as a parameter.

This is a common refactoring pattern, and so is worth learning well. First we give the code, and then the explanation.

```
var copy_attributes_factory = function(keys){

    return function(src, tgt){
        var i;
        var key;

        for(i=0; i< keys.length; i++){
            tgt[key] = src['get_' + key]();
        };
    };
};
```

Here's what we've done. We created a wrapper function, with *keys* as its only parameter. This will be our factory function. We've placed the original function into the body of the wrapper, and return it (as the value of the wrapper function). Finally, we clean up by removing the *keys* constant from the inner function.

There, it's done! When the factory function executes it returns an instance of the inner function for which *keys* is the argument we supplied to the factory.

A prototype class

The third task is to write a Fields class that has a copy_attributes method.

Here's a function prototype based solution. It relies on the user of the class supplying the *new* operator to create the new instance (and chaos results if the user forgets).

```
var Fields = function(keys){
    this.keys = keys;
};

Fields.prototype = {
    copy_attributes : function(src, tgt){

        var keys = this.keys;
        var i;

        for(i=0; i< keys.length; i++){
            tgt[key] = src['get_' + key]();
        };
    }
};
```

A minor point is the trailing sequence of right curly braces. Two are followed by a semicolon, but the middle one is not. Why? What happens if we put a semicolon there?

A SimpleClass solution

Here's another solution to the third task, which does not require the user to supply a *new* operator when creating instances. Instead it uses a factory function, which we call *Class* , that creates an instance constructor function from a prototype object.

```
// Rhino $ js -f library.js -f solution-2-d.js -

var fields = {};

fields.__init__ = function(keys){
    this.keys = keys;
};

fields.copy_attributes = function(src, tgt){

    var keys = this.keys;
    var key;
    var i;

    for(i=0; i< keys.length; i++){
        key = keys[i];
        tgt[key] = src['get_' + key]();
    };
};

var Fields = SimpleClass(fields);
```

Exercise: write test

Here's an exercise. Write a JavaScript file that tests this solution.

Exercise: bound method class

Recall that this-based instance methods are somewhat fragile. In other words, code like this will fail (and make unwanted changes to the global object):

```
var instance = MyClass(arg1, arg2);

var fn = instance.method;
element.onclick(fn);
```

Let's say that a class *has bound methods* if we can safely pass around instance methods. In other words, we want *instance.method* to have no dependence on the value of *this*.

We can achieve this by using a different class factory function.

```
var myclass = {}    // The prototype object.
// Add methods to myclass.
var MyClass = BoundMethodClass(myclass);
```

The exercise is to create (and test) a BoundMethodClass factory.