

# JavaScript Writ Large

Douglas Crockford

Yahoo! Inc.

The World's Most Popular  
Programming Language

The World's Most Unpopular  
Programming Language

JavaScript was supposed to be a HyperTalk-like language.

The goal for Netscape Navigator 2 was to provide HyperCard-like functionality in a web browser.

When Ajax gave JavaScript  
second chance, it succeeded  
because the language works.

The language has very good parts, and very bad parts.

By subsetting the language, you get a much better language.

JavaScript is not like any other  
language you have ever used.

Probably.

So you need to adapt.

The language can handle  
projects of significant size and  
complexity.

How much pain you will have to  
endure will depend on your  
technique.

The unit of pain is the agon.

# JavaScript Agon Reduction

1. Adopt a rigorous style
2. Avoid classical patterns
3. Think functionally
4. Think prototypally
5. Beware the DOM
6. Manage the divide

# 1. Adopt a rigorous style

- Bother to learn the language first.
- Use JSLint to constrain yourself to the Good Parts.
- Avoid global variables.

*Unearthing the excellence in JavaScript*



# JavaScript: The Good Parts

O'REILLY®

YAHOO! PRESS

*Douglas Crockford*

# JSLint

- JSLint defines a professional subset of JavaScript.
- It imposes a programming discipline that makes me much more confident in a dynamic, loosely-typed environment.
- <http://www.JSLint.com/>

# WARNING!

JSLint will hurt your  
feelings.

# Avoid global variables

- Global variables are evil.
- Lacking any form of linker, JavaScript uses global variables to bind separate compilation units together.
- Variables are a global by default!

# Define at most one global variable per compilation unit

- Use the global space as though it is package space.
- Do not put any ordinary variables in global space.
- Instead put whole objects in global space.
- Write global variables all in UPPERCASE!

# A single container in global space

```
var MYAPP = {  
    important_variable: true,  
    key_method: function (...) {  
        ...  
    }  
};
```

## 2. Avoid classical patterns

- JavaScript is not Java.
- It is class-free.
- It is powerful enough to simulate a classical style.
- It likes very shallow hierarchies.
- It is dynamic.
- It is loosely typed.

If you find yourself needing  
**super** methods, then you  
aren't ready yet.

# 3. Think functionally

Functions are used as

- Functions
- Methods
- Constructors
- Classes
- Modules

# Functions are first-class values

- Functions can be stored in variables, in objects (methods), passed as arguments, and returned.
- Functions can be defined inside of other functions with lexical (or static) scoping.
- A function has access to the variables and parameters of the functions it is contained within.

# Global

```
var names = ['zero', 'one', 'two',  
            'three', 'four', 'five', 'six',  
            'seven', 'eight', 'nine'];
```

```
var digit_name = function (n) {  
    return names[n];  
};
```

```
alert(digit_name(3));    // 'three'
```

# Slow

```
var digit_name = function (n) {  
    var names = ['zero', 'one', 'two',  
                'three', 'four', 'five', 'six',  
                'seven', 'eight', 'nine'];  
  
    return names[n];  
};  
  
alert(digit_name(3));    // 'three'
```

# Closure

```
var digit_name = (function () {  
    var names = ['zero', 'one', 'two',  
                'three', 'four', 'five', 'six',  
                'seven', 'eight', 'nine'];  
  
    return function (n) {  
        return names[n];  
    };  
})();  
  
alert(digit_name(3));    // 'three'
```

# Use functions to make functions

```
var make_add_x = function (x) {  
    return function (y) {  
        return x + y;  
    };  
};  
  
var add_2 = make_add_x(2);  
alert(add_2(7)); // 9
```

# Use functions to make functions

```
var make_f_x = function (f, x) {  
    return function (y) {  
        return f(x, y);  
    };  
};  
  
var add_2 = make_f_x(add, 2);  
alert(add_2(7)); // 9
```

# Fibonacci

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

- `fibonacci(40)`
- Calls itself 331,160,280 times.

# Memoizer

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

# Memoizer

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

- `fibonacci(40)`
- Calls itself 38 times.

# Memoizer

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

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

# Use functions to make objects

1. Make an object.
  - Object literal
  - `new`
  - `Object.create`
  - call another constructor  
(Functional Inheritance)

# Use functions to make objects

1. Make an object.
  - Object literal, `new`, `Object.create`, call another constructor
2. Define some variables and functions.
  - These become private members.

# Use functions to make objects

1. Make an object.
  - Object literal, `new`, `Object.create`, call another constructor
2. Define some variables and functions.
  - These become private members.
3. Augment the object with privileged methods.

# Use functions to make objects

1. Make an object.
  - Object literal, `new`, `Object.create`, call another constructor
2. Define some variables and functions.
  - These become private members.
3. Augment the object with privileged methods.
4. Return the object.

# Step One

```
function myPowerConstructor(x) {  
    var that = otherMaker(x);  
}
```

# Step Two

```
function myPowerConstructor(x) {  
    var that = otherMaker(x);  
    var secret = f(x);  
}
```

# Step Three

```
function myPowerConstructor(x) {  
  var that = otherMaker(x);  
  var secret = f(x);  
  that.priv = function () {  
    ... secret x ...  
  };  
}
```

# Step Four

```
function myPowerConstructor(x) {  
  var that = otherMaker(x);  
  var secret = f(x);  
  that.priv = function () {  
    ... secret x ...  
  };  
  return that;  
}
```

# Use functions to make objects

- Objects made with this pattern are robust and tamper-proof.
- We make objects with private state.
- There is a cost of one function object per method per instance.
- Ideal if the number of instances is fewer than hundreds.

## 4. Think prototypally

- Objects can inherit directly from other objects.
- Delegation, or differential inheritance.
- In creating an object, you only need to specify the properties that differ from its prototype.
- Objects can be customized with simple assignment.
- No classes!

# Delegation

- Every object contains a hidden property that contains a reference to the object's prototype.
- The chain always ends with `Object.prototype`.
- When fetching a property from an object, if the object lacks a property with that name, then its prototype is searched and then its prototype is searched...

# Prototype

- A popular pattern is to put common methods into a prototype object.
- There is then no memory or construction cost for providing methods to instances through the prototype.
- This is ideal when the number of instances is thousands or more.
- There is no privacy. All members are public.

## 5. Beware of the DOM

- If JavaScript were sped up to be infinitely fast, most applications would run at about the same speed.
- The DOM is a terrible bottleneck. HTML is an inefficient display list.
- When making large changes to the display, `innerHTML` is significantly faster than the DOM API.

## 6. Manage the divide

- The client and server are in a dialog.
- Make the messages between them as small as possible.
- The client does not need a copy of the database. It just needs, at any moment, just enough information to serve the user.
- Don't rewrite the server application in JavaScript!

# Division of Labor

How is the application divided  
between the browser and the  
server?

# Pendulum of Despair



Server

The browser is a terminal.

# Pendulum of Despair



Server  
Browser

The browser is a terminal

The server is a file system.

# Seek the Middle Way.

A pleasant dialogue between  
specialized peers.

# JavaScript Agon Reduction

1. Adopt a rigorous style
2. Avoid classical patterns
3. Think functionally
4. Think prototypally
5. Beware the DOM
6. Manage the divide

# The World's Most Misunderstood Programming Language