JavaScript: The Good Parts

Douglas Crockford

Yahoo! Inc.

http://www.crockford.com/codecamp/

The World's Most Popular Programming Language

The World's Most Popular Programming Language

The World's Most Unpopular Programming Language

A language of many contrasts.

The broadest range of programmer skills of any programming language.

From computer scientists
to cut-n-pasters
and everyone in between.

It is the language that people use without bothering to learn it first.

Complaints

- "JavaScript is not a language I know."
- "The browser programming experience is awful."
- "It's not fast enough."
- "The language is just a pile of mistakes."

Hidden under a huge steaming pile of good intentions and blunders is an elegant, expressive programming language.

JavaScript has good parts.

JavaScript is succeeding very well in an environment where Java was a total failure.

Influences

- Self
  - prototypal inheritance
  - dynamic objects
- Scheme
  - lambda
  - loose typing
- Java
  - syntax
  - conventions
- Perl
  - regular expressions

Bad Parts

- Global Variables
- + adds and concatenates
- Semicolon insertion
- typeof
- with and eval
- phony arrays
- == and !=
- false, null, undefined, NaN

Transitivity? What's That?

```
• '' == '0'           // false
• 0 == ''             // true
• 0 == '0'            // true
• false == 'false'    // false
• false == '0'        // true
• false == undefined  // false
• false == null       // false
• null == undefined   // true
• " \t\r\n " == 0     // true
```

```
value = myObject[name];
if (value == null) {
    alert(name + ' not found.');
}
```

Two errors that cancel each other out.

```
value = myObject[name];
if (value === undefined) {
    alert(name + ' not found.');
}
```

Good features that interact badly

- Objects can inherit from other objects.
- Functions can be members of objects.
- for..in statement mixes inherited functions with the desired data members.

for in is troublesome

- Design question: Should for..in do a shallow skim or a deep dredge?
- Decision: Deep dredge. The programmer must explicitly filter out the deep members.
- Except: They didn't tell anybody!
- Consequence: Lots of confusion about how to use for..in.

for in is troublesome

- Better Decision: Don't release the language broadly until we have enough experience to have confidence that we made the right choice.
- Historical Context: Getting it right at Netscape wasn't an option.

Bad Heritage

- Blockless statements

```
if (foo)
    bar();
```

- Expression statements

```
foo;
```

- Floating point arithmetic

```
0.1 + 0.2 !== 0.3
```

- ++ and --
- switch

Good Parts

- Lambda
- Dynamic Objects
- Loose Typing
- Object Literals

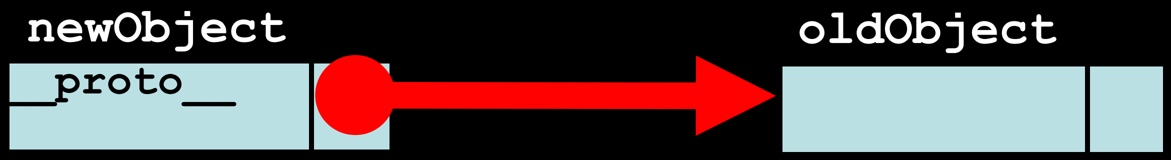
Inheritance

- Inheritance is object-oriented code reuse.
- Two Schools:
  - Classical
  - Prototypal

Prototypal Inheritance

- Class-free.
- Objects inherit from objects.
- An object contains a link to another object: Delegation. Differential Inheritance.

```
var newObject =
        Object.create(oldObject);
```



Prototypal Inheritance

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

new

- The **`new`** operator is required when calling a Constructor function.
- If **`new`** is omitted, the global object is clobbered by the constructor.
- There is no compile-time or run-time warning.

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'
```

A Module Pattern

```
var singleton = (function () {
    var privateVariable;
    function privateFunction(x) {
        ...privateVariable...
    }
    return {
        firstMethod: function (a, b) {
            ...privateVariable...
        },
        secondMethod: function (c) {
            ...privateFunction()...
        }
    };
}());
```

Module pattern is easily transformed into a powerful constructor pattern.

Power Constructors

1. Make an object.
   - Object literal
   - **`new`**
   - **`Object.create`**
   - call another power constructor

Power Constructors

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

Power Constructors

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

Power Constructors

1. Make an object.

- Object literal, **`new`**, **`Object.create`**, call another power constructor

1. Define some variables and functions.

- These become private members.

1. Augment the object with privileged methods.

2. 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 that ...
    };
}
```

Step Four

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

Closure

- A function object contains

  A function (name, parameters, body)

  A reference to the environment in which it was created (context).

- This is a very good thing.

Style Isn't Subjective

```
block
{
        ....
}
```

- Might work well in other languages

```
block {
        ....
}
```

- Works well in JavaScript

Style Isn't Subjective

```
return

{

    ok: false

};
```

- SILENT ERROR!

```
return {

    ok: true

};
```

- Works well in JavaScript

Style Isn't Subjective

```
return
{
    ok: false
};
```

Style Isn't Subjective

```
return; // semicolon insertion
{
    ok: false
};
```

Style Isn't Subjective

```
return;
{ // block
    ok: false
};
```

Style Isn't Subjective

```
return;
{
    ok: false  // label
};
```

Style Isn't Subjective

```
return;
{                       // useless
    ok: false           // expression
};                      // statement
```

Style Isn't Subjective

```
return;
{
    ok: false; // semicolon
};             // insertion
```

Style Isn't Subjective

```
return;
{
    ok: false;
}; // empty statement
```

Style Isn't Subjective

```
return;
{   // unreachable statement
    ok: false;
}
```

Style Isn't Subjective

```
return
{
    ok: false
};
```

- Bad style

```
return;
{
    ok: false;
}
```

- Bad results

Working with the Grain

A Personal Journey

Beautiful Code

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.

Unlearning Is Really Hard

Perfectly Fine == Faulty

It's not ignorance does so much damage; it's knowin' so derned much that ain't so.

Josh Billings

The Very Best Part:

Stability

No new design errors

since 1999!

Coming Soon

- [ES3.1] ECMAScript Fifth Edition
- Corrections
- Reality
- Support for object hardening
- Strict mode for reliability

```
"use strict";
```

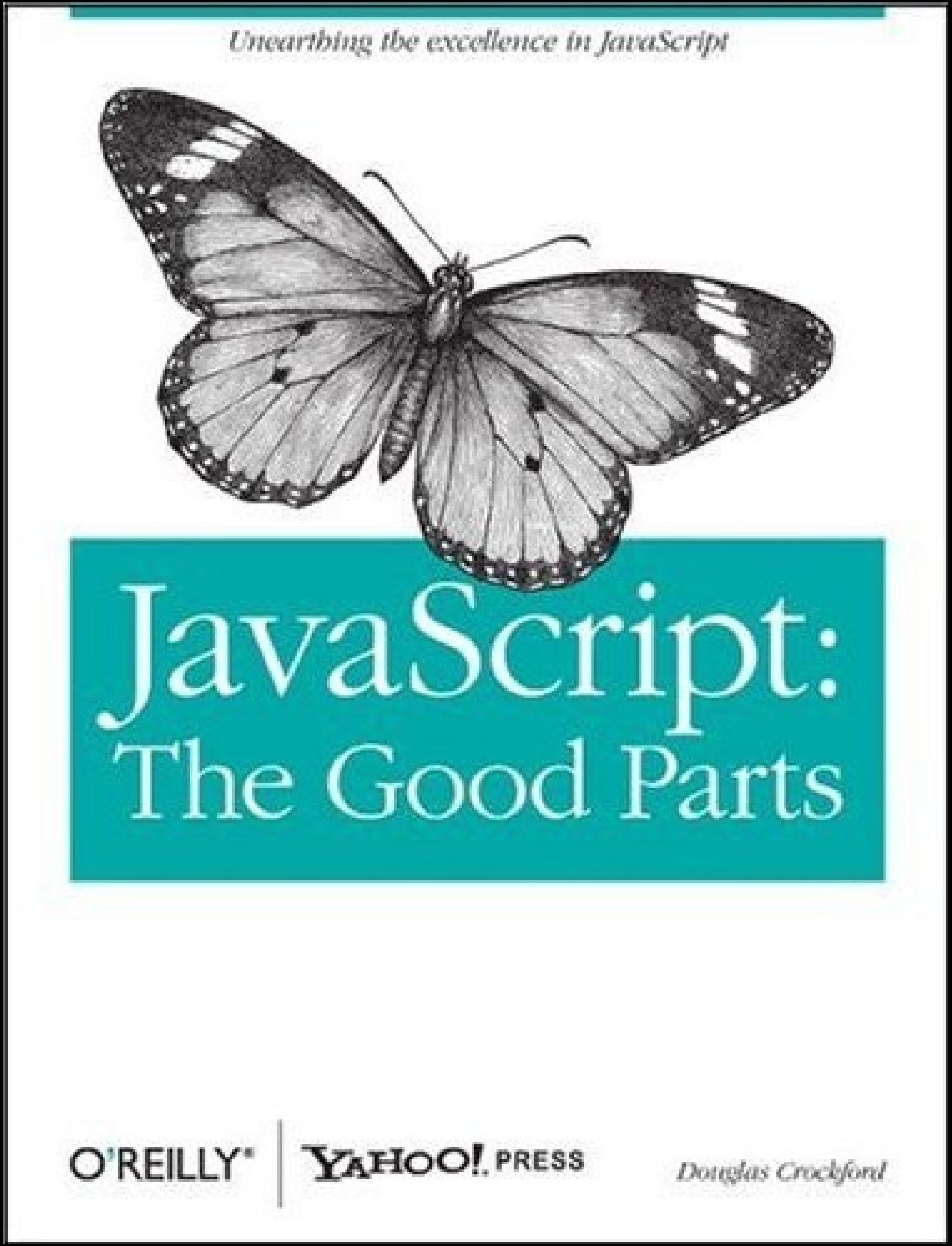
- Waiting on implementations

Safe Subsets

- The most effective way to make this language better is to make it smaller.
- FBJS
- Caja & Cajita
- Web Sandbox
- ADsafe
- The next edition of ECMAScript might include a secure formal subset.

The Good Parts

- Your JavaScript application can reach a potential audience of billions.
- If you avoid the bad parts, JavaScript works really well. There is some brilliance in it.
- It is possible to write good programs with JavaScript.



Unearthing the excellence in JavaScript

JavaScript:
The Good Parts

O'REILLY® | YAHOO!. PRESS

Douglas Crockford