JavaScript: The Good Parts
Douglas Crockford
Yahoo! Inc.
The World's Most Misunderstood Programming Language
The broadest range of programmer skills of any programming language.

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

- Java
  - syntax
  - conventions
- Self
  - prototypal inheritance
  - dynamic objects
- Scheme
  - lambda
  - loose typing
Bad Parts

• Global Variables
• + adds and concatenates
• Semicolon insertion
• typeof
• with and eval
• phony arrays
• for..in
• == and !=
• false, null, undefined, NaN
value = myObject[name];
if (value == null) {
    alert(name + ' not found.');
}

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

Bad Heritage

- Blockless statements
  ```javascript
  if (foo)
    bar();
  ```
- Expression statements
  ```javascript
  this.foo;
  ```
- Floating point arithmetic
  ```javascript
  0.1 + 0.2 !== 0.3
  ```
- switch
- ++ and --
Good Parts

- Lambdas
- Dynamic Objects
- Loose Typing
Inheritance

• Inheritance is object-oriented code reuse.

• Two Schools:
  • Classical
  • Prototypal
Prototypal Inheritance

- Class-free.
- Objects inherit from objects.
- An object contains a secret link to another object.

```javascript
var newObject = oldObject._prototype_.begetObject();
```
Prototypal Inheritance

```javascript
Object.prototype.begetObject =

    function () {
        function F() {}

        F.prototype = this;

        return new F();
    }
```
Object.prototype.begetObject = function () {
    function F() {}
    F.prototype = this;
    return new F();
}

newobject = oldobject.begetObject();

F
Object.prototype.begetObject = function () {
    function F() {} 
    F.prototype = this;
    return new F();
}

newobject = oldobject.begetObject();

F

prototype

constructor

oldobject
Object.prototype.begetObject = function () {
    function F() {}
    F.prototype = this;
    return new F();
}

newobject = oldobject.begetObject();
begetObject method

Object.prototype.begetObject = function () {
    function F() {}
    F.prototype = this;
    return new F();
}

newobject = oldobject.begetObject();
new

• The `new` operator is **required** when calling a Constructor.

• If `new` is omitted, there is no compile-time or run-time warning.

• The global object is clobbered by the constructor.
A Module Pattern

```javascript
var singleton = function () {
    var privateVariable;

    function privateFunction(x) {
        ...privateVariable...
    }

    return {
        firstMethod: function (a, b) {
            ...privateVariable...
        },
        secondMethod: function (c) {
            ...privateFunction()...
        }
    };

})();
```
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.
later method

- The \texttt{later} method causes a method on the object to be invoked in the future.

\texttt{my\_object.later(1000, "erase", true);}
later method

Object.prototype.later =
    function (msec, method) {
        var that = this;
        var args = Array.prototype.slice.
                    apply(arguments, [2]);
        if (typeof method === 'string') {
            method = that[method];
        }
        setTimeout(function () {
            method.apply(that, args);
        }, msec);
        return that;
    }
Event Reg

myObject.

on('ready', beginProc).
on('busy', reschedule, [a, b]).
on('delete', 'erase');

myObject.fire({type: 'ready'});
```
function eventreg(o) {
    var handle = {};
    o.on = function (type, method, parameters) {
        var e = {
            method: method,
            parameters: parameters
        };
        if (handler[type]) {
            handler[type].push(e);
        } else {
            handler[type] = [e];
        }
        return o;
    };
    o.fire = function (event) {...};
    o.off = function (type, method) {...};
    return o;
}
```
o.fire = function (event) {...};

var e, // handler record
    f, // handler function
    i, // loop index
    h = handler[m.type]; // array of handler records

if (handler) {
    for (i = 0; i < h.length; i += 1) {
        e = h[i];
        f = e.method;
        if (isString(f)) {
            f = o[f];
        }
        f.apply(this, e.parameters || [event]);
    }
}

return o;
Inheritance Patterns

• Prototypal Inheritance works really well with public methods.

• Parasitic Inheritance works really well with privileged and private and public methods.

• Pseudoclassical Inheritance for elderly programmers who are old and set in their ways.
Working with the Grain

• Pseudoclassical patterns are less effective than prototypal patterns or parasitic patterns.

• Formal classes are not needed for reuse or extension.

• Be shallow. Deep hierarchies are not effective.
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
Style Isn't Subjective

block {
    ....
}

block {
    ....
}

- Works well in JavaScript
- Might work well in other languages
Fixing JavaScript

• Deprecate the weak features.

• Fix the blunders carefully.

• Add new features that do not break syntax.

• Keep it simple. Keep it safe.

• Make it simpler. Make it safer.
Fixing JavaScript

- toJSONString and parseJSON
- a safe eval method
- object.dontEnum(name)

- No experiments.
- No radical changes.
The Very Best Part: Stability

No new design errors since 1999!
More Languages!

- The world is full of programming languages. Why restrict ourselves to just JavaScript?
- We need a classical Ajax language for programmers without the mental capacity to master JavaScript.
More Languages!

- We need a secure programming language.
- I believe it is possible to make a capability secure, JavaScript-like language.
- JavaScript will never be that language.
JavaScript

• It is a really good language if you avoid its weaknesses.

• Don't destabilize the language.

• Let's make new languages.

• This time without so many bad parts.