The JavaScript Programming Language

Douglas Crockford

Overview

- History
- Language
- Advanced Features
- Platforms
- Standards
- Style

The World's Most Misunderstood Programming Language

Sources of Misunderstanding

- The Name
- Mispositioning
- Design Errors
- Bad Implementations
- The Browser
- Bad Books
- Substandard Standard
- JavaScript is a Functional Language

History

- 1992
  - Oak, Gosling at Sun & FirstPerson
- 1995
  - HotJava
  - LiveScript, Eich at Netscape
- 1996
  - JScript at Microsoft
- 1998
  - ECMAScript

Not a Web Toy

- It is a real language
- Small, but sophisticated
- It is not a subset of Java

Key Ideas

- Load and go delivery
- Loose typing
- Objects as general containers
- Prototypal inheritance
- Lambda
- Linkage though global variables

Values

- **Numbers**
- **Strings**
- **Booleans**
- **Objects**
- `null`
- `undefined`

Numbers

- **Only one number type**

  No integers

- **64-bit floating point**
- **IEEE-754 (aka “`Double`”)**
- **Does not map well to common understanding of arithmetic:**
- `0.1 + 0.2 = 0.30000000000000004`

NaN

- **Special number: Not a Number**
- **Result of undefined or erroneous operations**
- **Toxic: any arithmetic operation with `NaN` as an input will have `NaN` as a result**
- **`NaN` is not equal to anything, including `NaN`**

Number function

`Number(value)`

- Converts the value into a number.
- It produces `NaN` if it has a problem.
- Similar to + prefix operator.

parseInt function

```
parseInt(value, 10)
```

- Converts the value into a number.
- It stops at the first non-digit character.
- The radix (10) should be required.

```
parseInt("08") === 0
parseInt("08", 10) === 8
```

Math

- Math object is modeled on Java's Math class.
- It contains

`abs`	absolute value
`floor`	integer
`log`	logarithm
`max`	maximum
`pow`	raise to a power
`random`	random number
`round`	nearest integer
`sin`	sine
`sqrt`	square root

Strings

- Sequence of 0 or more 16-bit characters
  - UCS-2, not quite UTF-16
  - No awareness of surrogate pairs
- No separate character type
  - Characters are represented as strings with a length of 1
- Strings are immutable
- Similar strings are equal (==)
- String literals can use single or double quotes

String `length`

- **string**`.length`

- **The** `length` **property determines the number of 16-bit characters in a string.**

String function

```
String(value)
```

- **Converts value to a string**

String Methods

- `charAt`
- `concat`
- `indexOf`
- `lastIndexOf`
- `match`
- `replace`
- `search`
- `slice`
- `split`
- `substring`
- `toLowerCase`
- `toUpperCase`

Booleans

- `true`
- `false`

Boolean function

`Boolean(`**value**`)`

- **returns** `true` **if value is truthy**
- **returns** `false` **if value is falsy**
- **Similar to** `!!` **prefix operator**

`null`

- **A value that isn't anything**

`undefined`

- A value that isn't even that
- The default value for variables and parameters
- The value of missing members in objects

Falsy values

- `false`
- `null`
- `undefined`
- `""` **(empty string)**
- `0`
- `NaN`

- **All other values (including all objects) are truthy.**

`"0"` `"false"`

Everything Else Is Objects

Dynamic Objects

- **Unification of Object and Hashtable**
- `new Object()` **produces an empty container of name/value pairs**
- **A name can be any string, a value can be any value except** `undefined`
- **members can be accessed with dot notation or subscript notation**
- **No hash nature is visible (no hash codes or rehash methods)**

Loosely Typed

- Any of these types can be stored in an variable, or passed as a parameter to any function
- The language is not "untyped"

C

- JavaScript is syntactically a C family language
- It differs from C mainly in its type system, which allows functions to be values

Identifiers

- Starts with a letter or _ or $
- Followed by zero or more letters, digits, _ or $

- By convention, all variables, parameters, members, and function names start with lower case
- Except for constructors which start with upper case

- Initial _ should be reserved for implementations
- $ should be reserved for machines.

Reserved Words

abstract
boolean **break** byte
case catch char class const **continue**
debugger **default delete do** double
else enum export extends
false final **finally** float **for function**
goto
if implements import **in instanceof** int
interface
long
native **new null**
package private protected public
return
short static super **switch** synchronized
this throw throws transient **true try typeof**
var volatile **void**
while with

Comments

```
// slashslash line comment

/*
    slashstar
    block
    comment
*/
```

Operators

- **Arithmetic**

  `+  -  *  /  %`

- **Comparison**

  `==  !=  <  >  <=  >=`

- **Logical**

  `&&  ||  !`

- **Bitwise**

  `&  |  ^  >>  >>>  <<`

- **Ternary**

  `?:`

+

- Addition and concatenation
- If both operands are numbers,

  then

    add them

  else

    convert them both to strings

    concatenate them

```
'$' + 3 + 4 = '$34'
```

+

- **Unary operator can convert strings to numbers**

```
+"42" = 42
```

- **Also**

```
Number("42") = 42
```

- **Also**

```
parseInt("42", 10) = 42
```

```
+"3" + (+"4") = 7
```

/

- **Division of two integers can produce a non-integer result**

```
10 / 3 = 3.3333333333333335
```

== !=

- Equal and not equal
- These operators can do type coercion
- It is better to use === and !==, which do not do type coercion.

&&

- The guard operator, aka *logical and*
- If first operand is truthy
  then result is second operand
  else result is first operand
- It can be used to avoid null references

```
if (a) {
   return a.member;
} else {
   return a;
}
```

- can be written as

```
return a && a.member;
```

||

- **The default operator, aka *logical or***
- **If first operand is truthy**
  then result is first operand
  else result is second operand
- **It can be used to fill in default values.**

```
var last = input || nr_items;
```

- **(If `input` is truthy, then `last` is input, otherwise set `last` to `nr_items`.)**

!

- **Prefix *logical not* operator.**
- **If the operand is truthy, the result is `false`. Otherwise, the result is `true`.**
- **`!!` produces booleans.**

Bitwise

& | ^ >> >>> <<

- The bitwise operators convert the operand to a 32-bit signed integer, and turn the result back into 64-bit floating point.

Statements

- *expression*
- `if`
- `switch`
- `while`
- `do`
- `for`
- `break`
- `continue`
- `return`
- `try/throw`

Break statement

- Statements can have labels.
- Break statements can refer to those labels.

```
loop: for (;;) {
   …
      if (...) {
         break loop;
      }
   …
}
```

For statement

- **Iterate through all of the elements of an array:**

```
for (var i = 0; i < array.length; i += 1) {

        // within the loop,
        // i is the index of the current member
        // array[i] is the current element


}
```

For statement

- **Iterate through all of the members of an object:**

```
for (var name in object) {
    if (object.hasOwnProperty(name)) {

        // within the loop,
        // name is the key of current member
        // object[name] is the current value

    }
}
```

Switch statement

- Multiway branch
- The switch value does not need to a number. It can be a string.
- The case values can be expressions.

Switch statement

```
switch (expression) {
case ';':
case ',':
case '.':
    punctuation();
    break;
default:
    noneOfTheAbove();
}
```

Throw statement

```
throw new Error(reason);

throw {
    name: exceptionName,
    message: reason
};
```

Try statement

```
try {
    ...
} catch (e) {
    switch (e.name) {
    case 'Error':
        ...
        break;
    default:
        throw e;
    }
}
```

Try Statement

- **The JavaScript implementation can produce these exception names:**

```
'Error'
'EvalError'
'RangeError'
'SyntaxError'
'TypeError'
'URIError'
```

With statement

- **Intended as a short-hand**
- **Ambiguous**
- **Error-prone**
- **Don't use it**

```
with (o) {
    foo = null;
}
```

- `o.foo = null;`
- `foo = null;`

Function statement

```
function name(parameters) {
  statements;
}
```

Var statement

- Defines variables within a function.
- Types are not specified.
- Initial values are optional.

```
var name;
var nrErrors = 0;
var a, b, c;
```

Scope

- In JavaScript, {blocks} do not have scope.
- Only functions have scope.
- Vars defined in a function are not visible outside of the function.

Return statement

```
return expression;
```

- **or**

```
return;
```

- **If there is no *expression*, then the return value is `undefined`.**
- **Except for constructors, whose default return value is `this`.**

Objects

- Everything else is objects
- Objects can contain data and methods
- Objects can inherit from other objects.

Collections

- An object is an unordered collection of name/value pairs
- Names are strings
- Values are any type, including other objects
- Good for representing records and trees
- Every object is a little database

Object Literals

- Object literals are wrapped in { }
- Names can be names or strings
- Values can be expressions
- : separates names and values
- , separates pairs
- Object literals can be used anywhere a value can appear

Object Literals

```
var myObject = {name: "Jack B. Nimble",
'goto': 'Jail', grade: 'A', level: 3};
```

"name"	"Jack B. Nimble"
"goto"	"Jail"
"grade"	"A"
"level"	3

```
var theName = myObject.name;
var destination = myObject['goto'];
```

Maker Function

```
function maker(name, where, grade, level) {
    var it = {};
    it.name = name;
    it['goto'] = where;
    it.grade = grade;
    it.level = level;
    return it;
}

myObject = maker("Jack B. Nimble",
    'Jail', 'A', 3);
```

Object Literals

```
var myObject = {name: "Jack B. Nimble",
'goto': 'Jail', grade: 'A', format: {type:
'rect', width: 1920, height: 1080,
interlace: false, framerate: 24}};
```

Object Literals

```
var myObject = {
    name: "Jack B. Nimble",
    'goto': 'Jail',
    grade: 'A',
    format: {
        type: 'rect',
        width: 1920,
        height: 1080,
        interlace: false,
        framerate: 24
    }
};
```

Object Literals

```
myFunction({
    type: 'rect',
    width: 1920,
    height: 1080
});
throw {
    name: 'error',
    message: 'out of bounds'
};
```

Object Literals

```
function SuperDiv(width, height,
  left, top, zIndex, position,
  color, visibility, html,
  cssClass)

function SuperDiv(spec)
```

Object Augmentation

- New members can be added to any object by simple assignment

- There is no need to define a new class

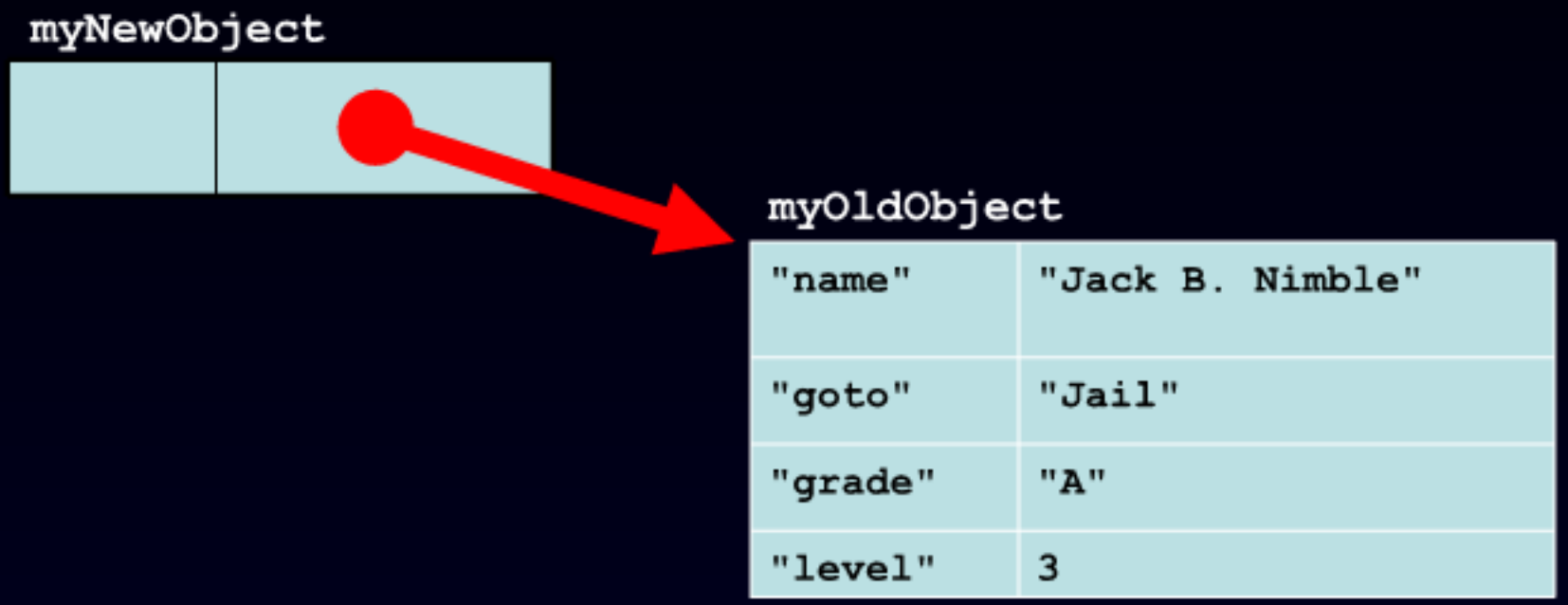
```
myObject.format.colorModel =
   'YCgCb';

myObject[name] = value;
```

Linkage

- Objects can be created with a secret link to another object.
- If an attempt to access a name fails, the secret linked object will be used.
- The secret link is not used when storing. New members are only added to the primary object.
- The `object(o)` function makes a new empty object with a link to object o.

Linkage

```
var myNewObject = object(myOldObject);
```



Linkage

```
myNewObject.name = "Tom Piperson";
myNewObject.level += 1;
myNewObject.crime = 'pignapping';
```

"name"	"Tom Piperson"
"level"	4
"crime"	"pignapping"

"name"	"Jack B. Nimble"
"goto"	"Jail"
"grade"	"A"
"level"	3

Inheritance

- Linkage provides simple inheritance.
- An object can inherit from an older object.

Prototypal Inheritance

- Some languages have classes, methods, constructors, and modules. JavaScript's functions do the work of all of those.
- Instead of Classical Inheritance, JavaScript has Prototypal Inheritance.
- It accomplishes the same things, but differently.
- It offers greater expressive power.
- But it's different.

Prototypal Inheritance

- Instead of organizing objects into rigid classes, new objects can be made that are similar to existing objects, and then customized.
- Object customization is a lot less work than making a class, and less overhead, too.
- One of the keys is the `object(o)` function.
- The other key is functions.

Object Methods

- **All objects are linked directly or indirectly to** `Object.prototype`
- **All objects inherit some basic methods.**
- **None of them are very useful.**
- `hasOwnProperty(`**name**`)`

  Is the name a true member of this object?
- **No** `copy` **method.**
- **No** `equals` **method.**

Object Construction

- Make a new empty object
- All three of these expressions have exactly the same result:

```
new Object()

{}

object(Object.prototype)
```

- `{}` is the preferred form.

Reference

- Objects can be passed as arguments to functions, and can be returned by functions
  - Objects are passed by reference.
  - Objects are not passed by value.

- The `===` operator compares object references, not values
  - `true` only if both operands are the same object

Delete

- **Members can be removed from an object with the `delete` operator**

```
delete myObject[name];
```

Arrays

- `Array` **inherits from** `Object`**.**
- **Indexes are converted to strings and used as names for retrieving values.**
- **Very efficient for sparse arrays.**
- **Not very efficient in most other cases.**
- **One advantage: No need to provide a length or type when creating an array.**

length

- **Arrays, unlike objects, have a special `length` member.**
- **It is always 1 larger than the highest integer subscript.**
- **It allows use of the traditional `for` statement.**

```
for (i = 0; i < a.length; i += 1) {
    ...
}
```

- **Do not use `for..in` with arrays**

Array Literals

- An array literal uses `[]`
- It can contain any number of expressions, separated by commas

```
myList = ['oats', 'peas', 'beans'];
```

- New items can be appended

```
myList[myList.length] = 'barley';
```

- The dot notation should not be used with arrays.
- `[]` is preferred to `new Array()`.

Array Methods

- concat
- join
- pop
- push
- slice
- sort
- splice

Deleting Elements

```
delete array[number]
```

- **Removes the element, but leaves a hole in the numbering.**

```
array.splice(number, 1)
```

- **Removes the element and renumbers all the following elements.**

Deleting Elements

```
myArray = ['a', 'b', 'c', 'd'];

delete myArray[1];

// ['a', undefined, 'c', 'd']

myArray.splice(1, 1);

// ['a', 'c', 'd']
```

Arrays v Objects

- Use objects when the names are arbitrary strings.
- Use arrays when the names are sequential integers.
- Don't get confused by the term Associative Array.

Distinguishing Arrays

```
value.constructor === Array
```

```
value instanceof Array
```

Neither of these work when the value comes from a different frame.

Arrays and Inheritance

- **Don’t use arrays as prototypes.**

  The object produced this way does not have array nature. It will inherit the array's values and methods, but not its `length`**.**

- **You can augment an individual array.**

  Assign a method to it.

  This works because arrays are objects.

- **You can augment all arrays.**

  Assign methods to `Array.prototype`

Functions

- Functions are first-class objects

3. Functions can be passed, returned, and stored just like any other value

5. Functions inherit from `Object` and can store name/value pairs.

Function operator

- The function operator takes an optional name, a parameter list, and a block of statements, and returns a function object.

```
function name(parameters) {
    statements
}
```

- A function can appear anywhere that an expression can appear.

lambda

- What JavaScript calls `function`, other languages call `lambda`.
- It is a source of enormous expressive power.
- Unlike most power-constructs, it is secure.

Function statement

- **The function statement is just a short-hand for a `var` statement with a function value.**

```
function foo() {}
```

expands to

```
var foo = function foo() {};
```

Inner functions

- Functions do not all have to be defined at the top level (or left edge).
- Functions can be defined inside of other functions.

Scope

- An inner function has access to the variables and parameters of functions that it is contained within.

- This is known as Static Scoping or Lexical Scoping.

Closure

- **The scope that an inner function enjoys continues even after the parent functions have returned.**

- **This is called *closure*.**

Example

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

Function Objects

- Functions are objects, so they can contain name/value pairs.
- This can serve the same purpose as `static` members in other languages.

Method

- Since functions are values, functions can be stored in objects.

- A function in an object is called a *method*.

Invocation

- If a function is called with too many arguments, the extra arguments are ignored.
- If a function is called with too few arguments, the missing values will be `undefined`.
- There is no implicit type checking on the arguments.

Invocation

- There are four ways to call a function:

  Function form

  functionObject(*arguments*)

  Method form

  thisObject.*methodName*(*arguments*)

  thisObject["*methodName*"](*arguments*)

  Constructor form

  new *functionObject*(*arguments*)

  Apply form

  functionObject.apply(*thisObject*,
  [*arguments*])

Method form

thisObject.*methodName*(*arguments*)

- When a function is called in the method form, `this` is set to *thisObject*, the object containing the function.
- This allows methods to have a reference to the object of interest.

Function form

functionObject(*arguments*)

- When a function is called in the function form, `this` is set to the global object.
  - That is not very useful.
  - It makes it harder to write helper functions within a method because the helper function does not get access to the outer `this`.

```
var that = this;
```

Constructor form

`new` ***functionObject*** `(`***arguments***`)`

- When a function is called with the `new` operator, a new object is created and assigned to `this`.
- If there is not an explicit return value, then `this` will be returned.

this

- `this` is an extra parameter. Its value depends on the calling form.
- `this` gives methods access to their objects.
- `this` is bound at invocation time.

Invocation form	this
function	the global object
method	the object
constructor	the new object

`arguments`

- **When a function is invoked, in addition to its parameters, it also gets a special parameter called `arguments`.**
- **It contains all of the arguments from the invocation.**
- **It is an array-like object.**
- **`arguments.length` is the number of arguments passed.**

Example

```
function sum() {
    var i,
        n = arguments.length,
        total = 0;
    for (i = 0; i < n; i += 1) {
        total += arguments[i];
    }
    return total;
}
```

Augmenting Built-in Types

- `Object.prototype`
- `Array.prototype`
- `Function.prototype`
- `Number.prototype`
- `String.prototype`
- `Boolean.prototype`

trim

```
String.prototype.trim = function () {
    return this.replace(
        /^\s*(\S*(\s+\S+)*)\s*$/, "$1");
};
```

supplant

```
var template = '<table border="{border}">' +
    '<tr><th>Last</th><td>{last}</td></tr>' +
    '<tr><th>First</th><td>{first}</td></tr>' +
    '</table>';

var data = {
    first: "Carl",
    last: "Hollywood",
    border: 2
};

mydiv.innerHTML = template.supplant(data);
```

supplant

```
String.prototype.supplant = function (o) {
    return this.replace(/{([^{}]*)}/g,
        function (a, b) {
            var r = o[b];
            return typeof r === 'string' ?
                r : a;
        }
    );
};
```

typeof

- **The `typeof` prefix operator returns a string identifying the type of a value.**

type	typeof
object	'object'
function	'function'
array	'object'
number	'number'
string	'string'
boolean	'boolean'
null	'object'
undefined	'undefined'

eval

`eval(`***string***`)`

- **The `eval` function compiles and executes a string and returns the result.**
- **It is what the browser uses to convert strings into actions.**
- **It is the most misused feature of the language.**

Function function

```
new Function(parameters, body)
```

- **The `Function` constructor takes zero or more parameter name strings, and a body string, and uses the JavaScript compiler to produce a function object.**
- **It should only be used to compile fresh source from a server.**
- **It is closely related to `eval`.**

Built-in Type Wrappers

- **Java has `int` and `Integer`, two incompatible types which can both carry the same value with differing levels of efficiency and convenience.**

- **JavaScript copied this pattern to no advantage. Avoid it.**
- **Avoid `new Boolean()`**
- **Avoid `new String()`**
- **Avoid `new Number()`**

Confession

```
function object(o) {
  function F() {}
  F.prototype = o;
  return new F();
}
```

Augmentation

- We can directly modify individual objects to give them just the characteristics we want.
- We can do this without having to create classes.
- We can then use our new object as the prototype for lots of new objects, each of which can also be augmented.

Working with the Grain

- Classical patterns are less effective than prototypal patterns or parasitic patterns.
- Formal classes are not needed for reuse or extension.

(global) Object

- **The object that dares not speak its name.**
- **It is the container for all global variables and all built-in objects.**
- **Sometimes `this` points to it.**

```
var global = this;
```

- **On browsers, `window` is the global object.**

Global variables are evil

- Functions within an application can clobber each other.
- Cooperating applications can clobber each other.
- Use of the global namespace must be minimized.

Implied Global

- Any var which is not properly declared is assumed to be global by default.
- This makes it easy for people who do not know or care about encapsulation to be productive, but it makes applications less reliable.
- JSLint is a tool which helps identify implied globals and other weaknesses.

http://www.JSLint.com

Namespace

- **Every object is a separate namespace.**
- **Use an object to organize your variables and functions.**
- **The `YAHOO` Object.**

```
<head>
<script>
YAHOO={};
</script>
```

- `http://twiki.corp.yahoo.com/view/Devel/TheYAHOOObject`

Encapsulate

- Function scope can create an encapsulation.
- Use an anonymous function to wrap your application.

Example

```
YAHOO.Trivia = function () {
    // define your common vars here
    // define your common functions here
    return {
        getNextPoser: function (cat, diff) {
            ...
        },
        showPoser: function () {
            ...
        }
    };
} ();
```

Thinking about type

- Trading type-safety for dynamism.
- JavaScript has no cast operator.
- Reflection is really easy, and usually unnecessary.
- Why inheritance?
  - Automatic casting
  - Code reuse
- Trading brittleness for flexibility.

Date

The `Date` function is based on Java's Date class.

It was not Y2K ready.

RegExp

- Regular expression pattern matcher
- Patterns are enclosed in slashes
- Example: a pattern that matches regular expressions

```
/\/(\\[^\x00-\x1f]|\[(\\[^\x00-\x1f]|[^\x00-
\x1f\\\/])*\]|[^\x00-\x1f\\\/\[])+\/[gim]*/
```

- Bizarre notation, difficult to read.

Threads

- The language definition is neutral on threads
- Some language processors (like SpiderMonkey) provide thread support
- Most application environments (like browsers) do not provide it
- Threads are evil

Platforms

- Browsers
- WSH and Dashboard
- Yahoo!Widgets
- DreamWeaver and Photoshop
- Embedded

ActionScript

- Empty strings are truthy
- keywords are case insensitive
- No Unicode support
- No `RegExp`
- No `try`
- No statement labels
- `||` and `&&` return booleans
- separate operators for strings and numbers

E4X

- Extensions to ECMAScript for XML
- Proposed by BEA
- Allows `<XML>` literals
- Not compatible with ECMAScript Third Edition
- Not widely accepted yet
- Not in IE7

ECMAScript Fourth Edition

- A very large set of new features are being considered.
- Mozilla and Opera are committed.
- It is not clear that Microsoft will adopt it.
- No word from Safari yet.

Style

- Programming style isn't about personal taste.
- It is about rigor in expression.
- It is about clearness in presentation.
- It is about product adaptability and longevity.
- Good rules help us to keep the quality of our programs high.

Style and JavaScript

- Style is critically important for JavaScript.
- The dynamic nature of the language is considered by some to be "too soft". Discipline is necessary for balance.
- Most of the world's body of JavaScript programs is crap.

Code Conventions for the JavaScript Programming Language

http://javascript.crockford.com/code.html

Semicolon insertion

- When the compiler sees an error, it attempts to replace a nearby linefeed with a semicolon and try again.
- This should alarm you.
- It can mask errors.
- Always use the full, correct forms, including semicolons.

Line Ending

- **Break a line after a punctuator:**

```
, . ; : { } ( [ = < > ? ! + - * / %
~ ^ | & == != <= >= += -= *= /= %=
^= |= &= << >> || && === !== <<= >>=
>>> >>>=
```

- **Do not break after a name, string, number, or** `) ] ++ --`

- **Defense against copy/paste errors.**

Comma

- **Avoid tricky expressions using the comma operators.**

- **Do not use extra commas in array literals.**

- **Good:** `[1, 2, 3]`
- **Bad:** `[1, 2, 3,]`

Required Blocks

- **Good:**

```
if (a) {
    b();
}
```

- **Bad:**

```
if (a) b();
```

Forbidden Blocks

- **Blocks do not have scope in JavaScript.**
- **Blocks should only be used with structured statements**

```
function
if
switch
while
for
do
try
```

Variables

- Define all variables at the beginning of the function.

- JavaScript does not have block scope, so their is no advantage in declaring variables at the place of their first use.

Expression Statements

- **Any expression can be used as a statement. That can mask errors.**

- **Only assignment expressions and invocation expressions should be used as statements.**

- **Good:**

```
foo();
```

- **Bad:**

```
foo && foo();
```

`switch` Statement

- **Avoid using fallthrough.**
- **Each clause should explicitly `break` or `return` or `throw`.**

Assignment Expressions

- **Do not use assignment expressions in the condition parts of `if`, `while`, or `for`.**

- **It is more likely that**

```
if (a = b) { ... }
```

- **was intended to be**

```
if (a == b) { ... }
```

- **Avoid tricky expressions.**

== and !=

- **Be aware that == and != do type coercion.**
- **Bad**

```
if (a == null) { ... }
```

- **Good:**

```
if (a === null) { ... }
if (!a) { ... }
```

Labels

- **Use labels only on these statements:**

  `do`

  `for`

  `switch`

  `while`

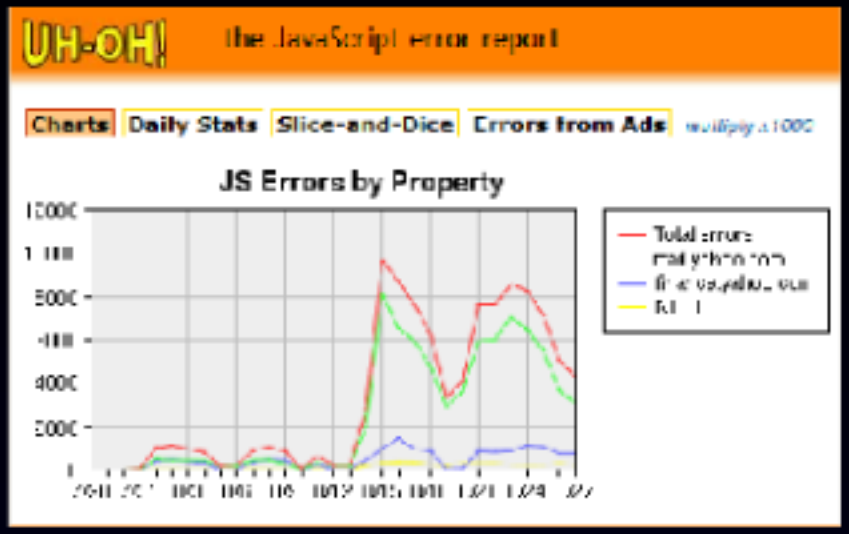
- **Never use** `javascript:` **as a label.**

JSLint

- JSLint can help improve the robustness and portability of your programs.
- It enforces style rules.
- It can spot some errors that are very difficult to find in debugging.
- It can help eliminate implied globals.
- Currently available on the web and as a Konfabulator widget.
- Soon, in text editors and Eclipse.

`http://www.JSLint.com/`

UHOH!



- **Universal Header Onerror Handler**
- **Inserted into 0.1% of pages**
- **Reports on JavaScript errors**
- `http://uhoh.corp.yahoo.com/`

Key Ideas

- Load and go delivery
- Loose typing
- Objects as general containers
- Prototypal inheritance
- Lambda
- Linkage though global variables

The JavaScript Programming Language

Douglas Crockford

crock@yahoo-inc.com

produce.yahoo.com/crock/javascript.ppt