Using JavaScript Well

Douglas Crockford
Yahoo!

The World's Most Misunderstood Programming Language

The World's Most Popular Programming Language

The World's Most Popular Programming Language

The World's Most Unpopular Programming Language

JavaScript was a failed language.

It should have died when Netscape died.

The language survived because of the Ajax Revolution [2005].

It is flourishing now because the language works.

JavaScript runs everywhere

- Web
- Mobile
- Embedded
- OS
- TV sets
- Databases (CouchDB, MongoDB)
- Servers

JavaScript Performance

- The initial implementations were very slow.
- Modern JavaScript engines are much faster.
- We don’t know how much faster because the benchmarks are bogus, and all implementers are gaming the benchmarks.
- Most applications are limited by DOM, not JavaScript.

Broadest range of skills

- Computer scientists.
- Cut-and-pasters.
- Everybody in between.
- Making a language that works for people at either extreme is difficult.
- Making a language that works for people at both extremes is amazing.

Bad Parts

- All languages contain some bad parts.
- JavaScript contains lots and lots of bad parts.
- JavaScript was designed and delivered to the world much too quickly.
- The problem with bad parts isn’t that they are useless. Bad parts are dangerous.
- Most of the bad parts can be avoided.

Good Parts

- By choosing to use only the good parts, you can use the excellent language that is hidden inside of the least popular language.
- JavaScript contains some of the best parts ever put into a programming language.

The language has so much expressive power that you can get a lot of work done without learning the language first.

Never program in ignorance.

By mastering the good parts, you can use the language well.

And now, the Good Parts.

Numbers

- A single binary floating point number type: IEEE 754

  Numbers are values virtually composed of thousands of bits.

  Only the 53 most significant bits are retained.

  These numbers are finite. But real numbers are not. So most rational numbers, and all irrational numbers, can only be approximated.

Associative law does not hold

- (a + b) + c is not guaranteed to always be the same as a + (b + c).
- Approximate values contain a small amount of noise.
- The amount of noise can accumulate over many calculations.
- The order of evaluation can have an impact on the amount of noise accumulated.

Integers are exact

- Integers have no noise and associate correctly as long as all values are less than 9007199254740992.
- Larger integers can be noisy.
- Most other languages contain int types in which this can be true:

```
a > 0, b > 0

(a + b) < 0   // insane
```

Decimal fractions are very noisy

- The most reported bug:

```
0.1 + 0.2 === 0.3   // false
```

- Binary floating point cannot exactly represent decimal fractions.
- This is only a problem on planets that use the decimal system
- Be very careful when adding people's money.

Strings

- Unicode (UCS-2)
- Literals

```
"text"

'symbol'

'cat' === 'c' + 'a' + 't'  // true
```

Booleans

- `true`
- `false`

Boolish values

- falsy

  ```
  false
  0
  NaN
  ""   // empty string
  null
  undefined
  ```

- truthy

  All other values including empty objects and empty arrays.

Objects

- Simple associative containers (hashtable, dictionary).
- Store and retrieve suffixes:

  object`[`expression`]`

  object`.`name
- Reflection is free

  Retrieving from a missing key produces the **`undefined`** value.
- **`Object.keys`**`(`object`)`

  produces an array containing all of the enumerable keys.

Object Literal

```
flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

Array

- Arrays are just objects with additional methods and a special property:

  `.length`

  The number of the largest positive element, plus one.

Array

```
var veggies = [
    "oats", "peas", "beans"
];
alert(veggies.length);    // 3
veggies[veggies.length] =
  "barley";
alert(veggies.length);    // 4
```

Everything is an object

Except `null` and `undefined`

Prototype

- `Object.create(object)`

  Create a new empty object that uses object as its prototype.

- Delegation

  If retrieve of a key fails, retrieve the key from the prototype.

- Differential Inheritance

  Objects inherit from other objects. Class-free.

Prototypal Inheritance

- More powerful than Classical Inheritance.
- Classification is not necessary.

  With classical inheritance, it is necessary to create a taxonomy of the object system at the beginning, usually before the system is well understood.

- Prototypal systems tend to be simpler, shallower, less ancestral coupling.

Function

- The best part.
  Like Scheme's lambda.
- Functions are first class objects.

- name
- (parameters)
- {body}

Statements

- Conditional
  - `if`
  - `switch`
- Loops
  - `while`
  - `do`
  - `for`
- **Disruption**
  - `break`
  - `return`
  - `try/throw`
- expression
  - assignment or function invocation

Operators

- Arithmetic

  `+ - * / %`

- Comparison

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

- Logical

  `&& || !`

- Bitwise

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

- Ternary

  `?:`

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'`

Array.isArray

Array.isArray(value) returns true if the value is an array.

var statement

- Private variables

  Explicitly declare all of the functions variables at the top of the body.

- Function scope

  The syntax promises blocks scope, but does not deliver.

- Free variables

  If a body does not declare a name as a variable or a parameter, then it can be obtained from an outer function.

- Closure

  A function can continue to access the variables of an outer function even after the outer function has returned.

`this`

- The **`this`** implicit parameter contains a reference to the object of invocation.
- **`this`** allows a method to know what object it is concerned with.
- **`this`** allows a single function object to service many functions.

Invocation

- The function ( ) suffix operator surrounding zero or more comma separated arguments.
- The arguments will be bound to parameters.

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.

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*`])`

Function form

functionObject (*arguments*)

- When a function is called in the function form, **`this`** is set to **`undefined`**.

Method form

thisObject`.`*methodName*`(`*arguments*`)`

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.

Constructor form

`new` *FunctionValue* (*arguments*)

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

Apply form

functionObject`.apply(`*thisObject, arguments*`)`

functionObject`.call(`*thisObject, argument…*`)`

- A function's **`apply`** or **`call`** method allows for calling the function, explicitly specifying *thisObject*.

`this`

- **`this`** is an bonus 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	**`undefined`**
method	the object
constructor	the new object
apply	argument

Worst Part: Global Object

- An application might be composed of several scripts.
- Each script is compiled and then executed in a common global environment.
- The global object is the container of all global variables.
- All global variables are shared.
- Accidental collisions are inevitable.
- This is one of the root causes of the XSS Attack.

Strict Mode

```
"use strict";
```

- Limits access to the global object.
- Makes it possible to have efficient and safe mashups.
- The most important new feature of ECMAScript Fifth Edition.
- IE9?

Those are the Good Parts.

Compose with the Good Parts to
make good programs.

JSLint

- JSLint is a code quaility tool for JavaScript, written in javaScript.
- JSLint defines a professional subset of JavaScript.
- JSLint understands the difference between Good Parts and Bad parts.
- Follow its advice.
- **`http://www.JSLint.com/`**

WARNING!

JSLint will hurt your feelings.

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'
```

```
function memoizer(memo, formula) {
    function recur(n) {
        return memo.hasOwnProperty(n) ?
            memo[n] :
            memo[n] = formula(recur, n);
    };
    return recur;
}

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

```
function memoizer(memo, formula) {
    function recur(n) {
        return memo.hasOwnProperty(n) ?
            memo[n] :
            memo[n] = formula(recur, n);
    };
    return recur;
}

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

```
function memoizer(memo, formula) {
    function recur(n) {
        return memo.hasOwnProperty(n) ?
            memo[n] :
            memo[n] = formula(recur, n);
    };
    return recur;
}

var digit_name = memoizer(['zero', 'one',
        'two', 'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine']);
```

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

Power Constructors

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

Power Constructors

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

Power Constructors

1. Make an object.
   - Object literal, `new`, `Object.create`, call another power 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 that ...
    };
}
```

Step Four

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

Thinking about performance

- Avoid premature optimization.
- Always measure before optimizing.
- Touch the DOM as lightly as possible.
- Avoid tuning for any particular browser.
- JavaScript and the DOM do not do well with huge data structures. Only ask for the data you need.

Thinking in functions

- Use the Good Parts to make objects and functions.
- Callbacks. Mixins. Chains. Compositions.
- There are lots of ways to get wonderful things to happen.