JavaScript: The Good Parts (Book Report)

Michal Dobrogost

November 18th, 2010

Introducing JavaScript

History

- Java$^{(tm)}$ applets failed, JavaScript took over
- Shunted from non-existance to world wide use
- Bad parts resulting from
  - Poor specification $\Longrightarrow$ poor portability
  - Difficult to read and modify code
  - Design mistakes
- “Beautiful, elegant, highly expressive language” buried inside

Introducing JavaScript

Features

Distinguishing features of JavaScript

- Dynamic typing
- First class functions
- Object literal notation
- Prototypal inheritance

Introducing JavaScript

Hello World!

```
-----------------------------------------
hello.html:
-----------------------------------------
<html><body>
<script src="hello.js">
</script>
</body></html>



-----------------------------------------
hello.js:
-----------------------------------------
document.writeln(’Hello world!’);
```

Syntax

Basics

- **Comments** : C-Style // or /* */ pairs
  - Do not use /* */ because they can occur in the program

```
/*
var x = /y*/.exec("yyyyyyy");
*/
```

- **Names** : letter (letter | digit | _)*
- **Numbers** : 23, 1.7, 1.1e-10, NaN, Infinity
- **Strings** : Unicode character set

```
"d" + ’o’ + "g" + ’\t’ === ’dog\t’
```

- **Boolean false** : false, null, undefined, ””, 0, NaN
- **Boolean true** : everything else

Syntax

Statements

- **Variable declaration**

  ```
  var x = 3, y = 1, z = 4;
  ```

- **If**

  ```
  if (x === 4) {
    <statements>
  } else {
    <statements>
  }
  ```

- **Switch**

  ```
  switch (x + y) {
    case 0:
    case 0 + 1:
      y = 3; // <statements>
      break  // <disruptive>
  }
  ```

Syntax

Loops

- **For**, **while** and **do while** loops similar to C

```
for (i = 0; i < 10; i += 1) {
  <statements>
}
```

- **For-in loop**

```
for (x in xs) {
  // Ensure x not from prototype chain
  if (xs.hasownProperty(x)) {
    ...
  }
}
```

Syntax

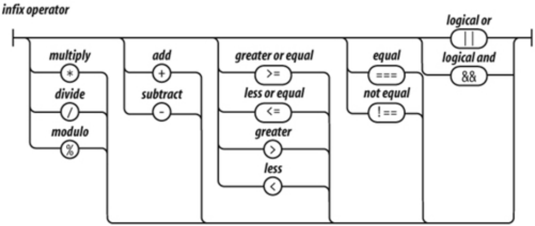
Exceptions, Functions

```
try {
  throw "ERROR";
}
catch (e) {
  if (e === "ERROR") {
    ...
  }
}

function f(x,y) {
  var z = x + y;
  return z;
}
```

Syntax

Expressions



Railroad diagram from [1].

- **prefix** : typeof, + (to number), - (negate), ! (logical not)
- **conditional** : $< expr >$? $< expr >$: $< expr >$
- **invocation** : function($< expr >, < expr >$)
- **refinement** : record.property, array[$< expr >$]

Syntax

Literals

Literals specify an object inline in the code.

- **object** : { name : ”bob”, ’age’ : 37 }
- **array** : [1,1,2,3,5,8]
- **regexp** : /[abc]*/ g i m

Syntax

Literals

Literals specify an object inline in the code.

- **object** : { name : ”bob”, ’age’ : 37 }
- **array** : [1,1,2,3,5,8]
- **regexp** : /[abc]*/ g i m
  - g ⟹ global, match multiple times
  - i ⟹ case insensitive
  - m ⟹ multiline

Objects

Defining, in detail

- Objects are passed by reference
- Reserved keywords are enclosed in ”” when used as properties of objects.

```
var delivery = {
  "for" : "Dr. Kahl",
  from : "Michal"
};
```

- Reserved keywords accessed by [”property”].

```
if (delivery.from === "Michal" &&
    delivery["for"] === "Dr. Kahl")
{
  ...
}
```

Objects

Non-existing fields

- Example

  ```
  var delivery = {
    "for" : "Dr. Kahl",
    from : "Michal"
  };
  ```

- Accessing non-existing property results in *undefined*.

  ```
  delivery.contents    // results in undefined
  ```

- Default values

  ```
  delivery.contents || "none"
  ```

- Guarded retrieval (prevent throwing TypeError)

  ```
  delivery.contents && delivery.contents.grade
  ```

Objects

Updates

- Example

```
var delivery = {
  "for" : "Dr. Kahl",
  from : "Michal"
};
```

- Change a property of the object

```
delivery.from = "Michal Dobrogost"
```

- Add a property to the object

```
delivery.contents = {
  material : "presentation.tex",
  grade : "A+"
}
```

- Removing a property from the object

```
delete delivery.contents.grade
```

Objects

Prototypical inheritance

- All objects subclass Object.prototype by default
- Prototype selected at creation time
- To sidestep complexities, Crockford suggests

```
if (typeof Object.beget !== 'function') {
  Object.beget = function(o) {
    var f = function () {};
    f.prototype = o;
    return new f();
  }
}
```

Objects

Inheritance example

```
var parent = { bye : "world" };
var child  = Object.beget(parent);
child.hi   = "Maggie";
parent.bye = "Ashley";

var str = ’Goodbye ’ + child.bye + ’, hello ’ + child.hi;
document.writeln(str + "!");
```

Prints...

Objects

Inheritance example

```
var parent = { bye : "world" };
var child  = Object.beget(parent);
child.hi   = "Maggie";
parent.bye = "Ashley";

var str = ’Goodbye ’ + child.bye + ’, hello ’ + child.hi;
document.writeln(str + "!");
```

Prints... “Goodbye Ashley, hello Maggie!”.

Objects

Reflection & Enumeration

```
for (prop in child) {
  document.writeln("prop:" + prop);
  document.writeln("own:" + child.hasOwnProperty(prop));
  document.writeln("type:" + (typeof child[prop]));
  document.writeln("val:" + child[prop]);
}

// prop:hi                 other types: number
// own:true                             string
// type:string                          object
// val:Maggie                           function
//                                      undefined
// prop:bye
// own:false
// type:string
// val:Ashley
```

Objects

Namespaces

No module or namespace facilities $\Longrightarrow$ fake it!

```
// Single global variable, our "namespace"
var NAMESP = {}

// All others are properties
NAMESP.parent = { bye : "world" };
NAMESP.child  = Object.beget(NAMESP.parent);
NAMESP.child.hi = "Maggie";
NAMESP.parent.bye = "Ashley";
```

Functions

as Methods

- Functions that are properties of objects are methods
- All functions take an implicit 'this' argument
    - Bound at invocation time.
- For methods 'this' binds to the containing object

```
var counter = {
  val : 0,
  inc : function () {
    this.val += 1;
  }
};
```

Functions

as Functions

- Functions that are not methods bind 'this' to the global object
- Crockford: This is a design mistake for inner functions
  - Should bind to the invoking function's 'this'

```
var counter = {
  val : 0,
  inc : function (howmuch) {
    var incOnce = function () {
      this.val += 1;                    // Problem!
    }
    for (i = 0; i < howmuch; i += 1) {
      incOnce();
    }
  }
};
```

Functions

as Functions ('that' pattern)

- Solved by introducing a variable 'that'
- Inner function is closure ⟹ 'that' is visible

```
var counter = {
  val : 0,
  inc : function (howmuch) {
    var that = this;                     // Solved!
    var incOnce = function () {
      that.val += 1;                     // Solved!
    }
    for (i = 0; i < howmuch; i += 1) {
      incOnce();
    }
  }
};
```

Functions

as Constructors

- Functions invoked with 'new' serve as constructors
- Bind 'this' to the object being created

```
var MakeHello = function () {
  this.hello = "world";
}

var x = new MakeHello();

document.writeln("Hello " + x.hello + "!");
```

Functions

as Constructors (with inheritance)

- Functions subclass Function.prototype
    - prototype property sets prototype for created object
- Now we can understand the beget function

```
if (typeof Object.beget !== 'function') {
  Object.beget = function(o) {
    var f = function () {}; // returns new object
    f.prototype = o;
    return new f();
  }
}
```

Functions

as invoked by apply

- Functions have apply property
  - apply is a method of Function.prototype
  - Takes the object to bind as 'this'
  - Takes the an array of arguments

```
var sum = add.apply(null, [5,8]);
```

- Can be used to apply methods with 'this' bound differently

```
var sister = {
  msg : "I like blue"
  show : function () { document.writeln(this.msg); }
};

var brother = { msg : "I like yellow" };

sister.show.apply(brother);
```

Functions

Scope, closures and currying

- Despite C-like syntax, a block does not start a new scope
- Inner functions have access to variables in scope at definition

```
var add = function(x) {
  {
    var z = x;
  }
  return function (y) { return z + y; }
}

var inc = add(1);

document.writeln(inc(11));
```

Prints... 12

Augmenting types

We can extend the functionality of a whole class of objects.

- functions
- strings
- numbers
- regular expressions
- booleans

```
Number.prototype.toInteger = function () {
  return Math[this < 0 ? ’ceiling’ : ’floor’](this);
}
```

```
document.writeln( (-1.5).toInteger() );
```

Prints... -1

The Bad Parts

(i) Hello world, gone wrong [1]

```
if ([0] == false) { document.writeln('Hello'); }
if ([0])          { document.writeln('world'); }
```

Prints...

[1] Idea from http://jimbojw.com [3]

The Bad Parts

(i) Hello world, gone wrong [1]

```
if ([0] == false) { document.writeln(’Hello’); }
if ([0])          { document.writeln(’world’); }
```

Prints...

```
Hello
world
```

Because...

[1] Idea from http://jimbojw.com [3]

The Bad Parts

(i) Hello world, gone wrong [1]

```
if ([0] == false) { document.writeln('Hello'); }
if ([0])          { document.writeln('world'); }
```

Prints...

```
Hello
world
```

Because...

- Type conversion results in 0 === 0
- [0] is an object so it is not false

[1] Idea from http://jimbojw.com [3]

The Bad Parts

(ii)

```
function fib(x) {
  if (x <= 1)  { return x; }
  return
    fib(x - 1) + fib(x - 2);
}

document.writeln(fib(1000));
```

What happens?

The Bad Parts

(ii)

```
function fib(x) {
  if (x <= 1)  { return x; }
  return
    fib(x - 1) + fib(x - 2);
}

document.writeln(fib(1000));
```

What happens? Prints undefined
Because...

The Bad Parts

(ii)

```
function fib(x) {
  if (x <= 1)  { return x; }
  return
    fib(x - 1) + fib(x - 2);
}

document.writeln(fib(1000));
```

What happens? Prints undefined
Because... JavaScript sees

```
function fib (x) {
  if (x <= 0) { return 0; }
  return;
  fib (x - 1) + fib(x - 2);
}
```

The Bad Parts

(iii)

```
document.writeln('4' - 2);
document.writeln('4' + 2);
```

Prints...

The Bad Parts

(iii)

```
document.writeln(’4’ - 2);
document.writeln(’4’ + 2);
```

Prints...

```
2
42
```

Because...

The Bad Parts

(iii)

```
document.writeln('4' - 2);
document.writeln('4' + 2);
```

Prints...

```
2
42
```

Because...

- typeof '4' === 'string'
- So + acts as string concatenation

Douglas Crockford, *JavaScript: The Good Parts*. O'Reilly, 2008.

ECMA International, *ECMAScript Language Specification 5th edition* 2009-12. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

JimboJW *stackoverflow.com : Why does 2 == [2] in JavaScript?* 2009-11-14. http://stackoverflow.com/questions/1724255/why-does-2-2-in-javascript

THANK YOU