JavaScript Design Regrets

茶少 from 阿里巴巴

新浪微博 @哦胖茶

Java Legacy

The diktat from upper engineering management was that the language must “look like Java”.

-- Brendan Eich

Date Class

JavaScript's Math and Date objects are based on classes from Java 1.0

```
const d = new Date('2016-09-10')
d.getDate()    // 10
d.getYear()    // 116 (2016 - 1900)
d.getMonth()   // 8
```

But since JDK 1.1, most methods in Date class have been deprecated…

Auto-Semicolon-Insertion (ASI)

- A lot of programming languages allows omitting semicolons:
- Swift, Python, Mathematica, Ruby, Groovy…
- **Only JavaScript** has problems with it

Auto-Semicolon-Insertion (ASI)

Imperfect Grammar -> Restricted Production

```
// returns undefined
return
{
    status: true
};

// returns { status: true }
return {
    status: true
};
```

Auto-Semicolon-Insertion (ASI)

```
var a = function(x) { console.log(x) }
(function() {
    // do something
})()
```

Solution: semicolon-less style

- Prefix a line with `;` whenever it starts with `( [ + - /`
- ```
  var a = function(x) { console.log(x) }
  ;(function() {
      // do something
  })()
  ```
  ```

Outdated Features

Hey, it was the 90s!

-- Brendan Eich

Stateful RegExp Functions

```
const re = /foo/g
console.log(re.test('foo bar')) // true
console.log(re.test('foo bar')) // false
```

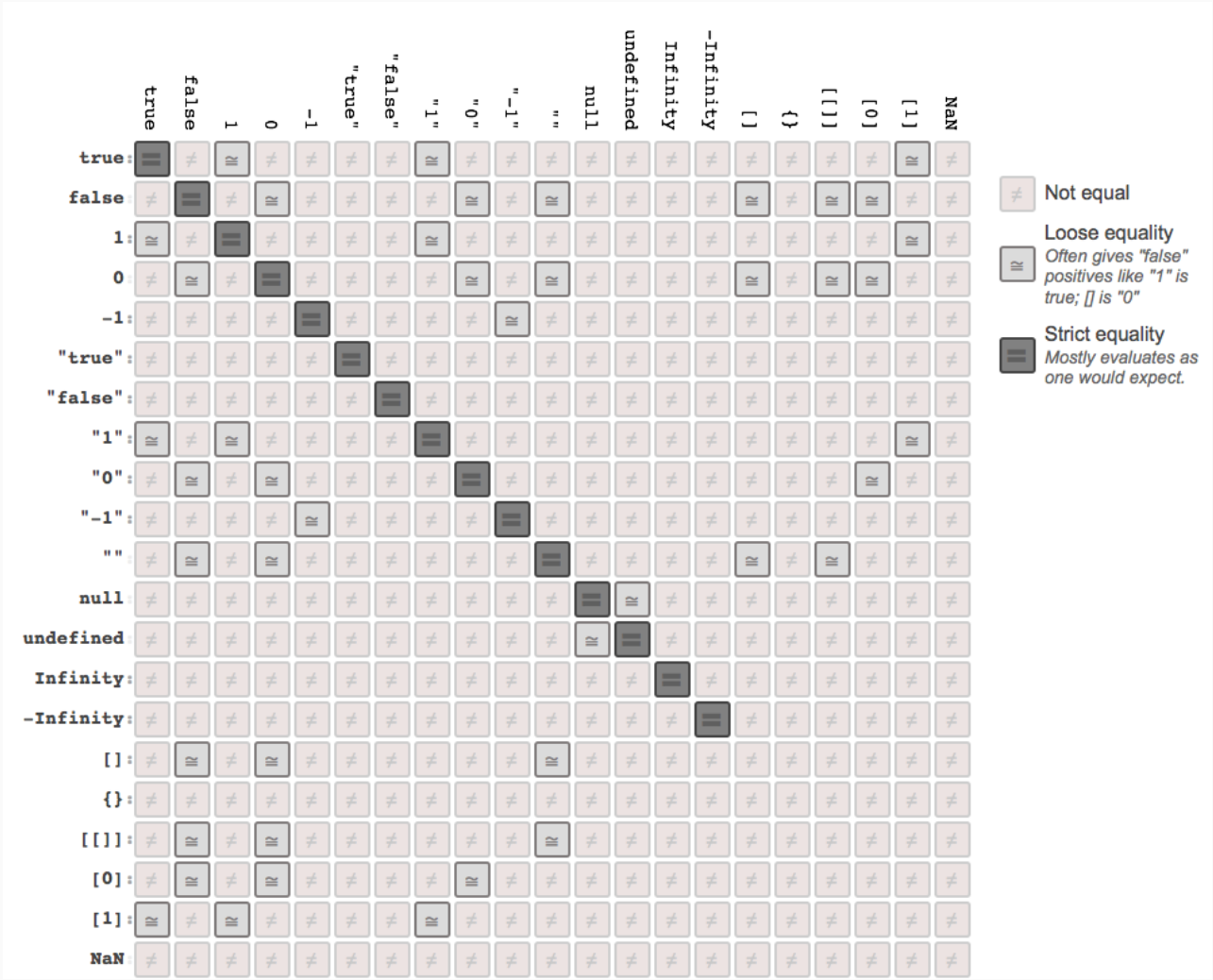
- Based on Perl 4
- No thread safety

Type System

Implicit Coercion

```
("foo" + + "bar") === "fooNaN" // true
'3' + 1 // '31'
'3' - 1 // 2
'222' - - '111' // 333
```

Equality Test


true
false
1
0
-1
"true"
"false"
"1"
"0"
"-1"
""
null
undefined
Infinity
-Infinity
[]
{}
[[]]
[0]
[1]
NaN
Not equal
Loose equality
Often gives "false" positives like "1" is true; [] is "0"
Strict equality
Mostly evaluates as one would expect.

Equality Test

```
[] == ![]                // true
3 == '3'                 // true

+0 === -0                // true
1 / +0 === 1 / -0        // false
NaN === NaN              // false
```

ES2015 Solution: Object.is

```
Object.is(NaN, NaN) // true
Object.is(+0, -0)   // false
```

Scope

Functional Scope

```
// The closure in loop problem
for (var i = 0; i !== 10; ++i) {
  // logs 10 ten times
  setTimeout(function() { console.log(i) }, 0)
}
```

Solution: Block Scope (let / const)

```
for (let i = 0; i !== 10; ++i) {
  // logs 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  setTimeout(function() { console.log(i) }, 0)
}
```

Variable Hoisting

```
bla = 2
var bla;
// ...


// is implicitly understood as:
var bla;
bla = 2;
```

Solution: Temporay Dead Zone

```
// without TDZ
console.log(a)  // undefined
var a = 1

// with TDZ
console.log(b)  // ReferenceError
let b = 2
```

Naming Regrets

There are only two hard things in Computer Science: cache invalidation and naming things.

-- Phil Karlton

let / const

- What const defines is not actual constant

- ```
  // const defines an immutable binding
  const MY_OBJECT = {"key": "value"};

  // Overwriting the object will cause TypeError
  MY_OBJECT = {"OTHER_KEY": "value"};

  // However, object keys are not protected,
  // so the following statement is executed without problem
  MY_OBJECT.key = "otherValue"; // Use Object.freeze() to make object immutable
  ```
  ```

let / const

If we had a “do-over”. I ‘d make let means what const now means and have something different for defining mutable lexical bindings. Maybe let var foo=...;.

-- Allen Wirfs-Brock (Project Editor and lead author of ES2015)

Function.prototype

- [[Prototype]] !== prototype
- fallbackOfObjectsCreatedWithNew `would be a better name`

Correct Ways to Access [[Prototype]]

- Object.getPrototypeOf() (ES5)
- .__proto__ (non-standard until ES2015, not supported in IE)

API Design Failure

import syntax

- JavaScript: `import { x, y } from z`
- Python: `from z import x, y`
- Actually, JavaScript's module system was inspired by Racket, not Python.
- The syntax would be much better if it's just copied from Python!

NaN

- isNaN

```
isNaN(123)          // false
isNaN(NaN)          // true
isNaN('a string');  // true
```

- Number.isNaN

Array Constructor Inconsistency

- `Array(1, 2, 3); // [1, 2, 3]`
- `Array(2, 3); // [2, 3]`
- `Array(3); // [,,] 黑人问号???`

Array-like Objects

```
typeof arguments.length     // number
Object.prototype.toString.call(arguments)    // [object Arguments]
arguments.slice(0, 1)       // TypeError: arguments.slice is not a function

var args = Array.prototype.slice.apply(arguments)
Object.prototype.toString.call(arguments)    // [object Array]
args.slice(0, 1)     // no error
```

Array-like Objects in ES2015

```
const nodeList = document.querySelectorAll('div')
const nodeArray = [...nodeList]

console.log(Object.prototype.toString.call(nodeList))    // [object NodeList]
console.log(Object.prototype.toString.call(nodeArray))    // [object Array]
```

Value Properties of the Global Object

NaN, Infinity, undefined are not reserved keywords

```
;(function(){
  var undefined = ‘foo’
  console.log(undefined, typeof undefined) // logs "foo string"
})()
```

eval

```
function test() {
  var x = 2, y = 4

  // Direct call, uses local scope, result is 6
  console.log(eval(“x + y”))

  var geval = eval;
  // Indirect call, uses global scope, throws ReferenceError because `x` is undefined
  console.log(geval("x + y"))
}
```

Why the difference?

- eval has access to local scope, which is dangerous
- In ES1, nobody cares
- In ES2, only direct call is allowed
- But it breaks the web
- So, browser vendors have to implement it to behave like new Function()

Q & A

Thanks!