JavaScript: The Good Parts

Part I

Agenda

- Introduction
- The Book
- JavaScript crash course and hands-on
  - Object Literals
  - Functions
  - Inheritance

The Book

amazon.com

Short, Exclusively about JavaScript

" Intended for programmers venturing into JavaScript for the first time "

" ..also intended for programmers who have been working with JavaScript at a novice level, and are now ready for a more sophisticated relationship with the language "

" This is not a book for beginners "
" This is not a book for dummies "

http://editorjs.heroku.com

(http://github.com/gnab/editorjs)

Object literal

- A collection of properties

```
var o = {
  key1: 'value1',
  key2: 13
};

o.key1          // 'value1'
o['key2']       // 13
o.key3          // undefined

delete o.key1
o.key1          // undefined
```

Array

02 Populating arrays

```
var array = [ 1, 2, 3 ];

array.push(4);

array[1]       // 2
array[4] = 5;
array.length   // 5
```

Object literal vs. Array (1/2)

- Arrays are not "real"
  - Objects in disquise
  - Operations are slow when number of elements are large

```
var obj = [ 1, 2, 3 ];
obj.speed = 5;

obj[1]          // 2
obj.speed       // 5
obj['speed']    // 5

obj.length      // 3
```

Object literal vs. Array (2/2)

- The for-in loops all properties of an object

```
var obj = [ 1, 2, 3 ];
obj.speed = 5;

var i;
for (i = 0; i < obj.length; i++) {
  obj[i];                    // 1,2,3
}

var key;
for (key in obj) {
  obj[key];                  // 1,2,3,5
}
```

Types


```
new Object()      // {}
new Array()       // []
new Number(1)     // 1
new Boolean(true) // true

typeof {}         // 'object'
typeof []         // 'object'     wtf?
typeof 1          // 'number'
typeof ''         // 'string'
typeof true       // 'boolean'

typeof null       // 'object'     wtf?
```

Functions

```
function () {}
```

Statements

```
function add (a, b) {
  return a + b;
}
```

```
var multiply = function (a, b) {
  return a * b;
};
```

Expressions

```
var recur = function rec (num) {
  if (num > 0) { rec(num--); };
  return num;
};
```

Functions
- Arguments

01 How many arguments do you have

- All functions can access their
  - arguments
    - length
    - callee

```
function countArgs () {
  return arguments.length;
}

countArgs()                 // 0
countArgs('one')            // 1
countArgs('one', 'two')     // 2
```

Functions
- First order variables

02 Higher order functions
03 Variables by reference

- Functions can return functions
- Functions can take functions as an argument

```
function createFunction () {
  return function () {

  };
}

var anonymous = createFunction();
anonymous() // undefined
```

Functions - Scope

- JavaScript has function scope!
  - Not block scope, like you may be used to

```
function () {

  i // undefined

  for (var i = 0; i < 10; i++) {

  }
  i // 10
}
```

wtf

Functions - Scope

- Variable declarations towards the top of your functions!

```
function () {
  var i;

  for (i = 0; i < 10; i++) {

  }
  i // 10
}
```

Functions - More on scope 1

- Functions can access everything from their outer scope

```
var a = 1;

function doSomething () {
  // a = 1

  function doMore () {
    // a = 1
  }
}
```

Functions - More on scope 2

- Not the other way around

```
function doSomething () {

  var b = 1;

  function doMore () {

    // b = 1
  }
}

// b = undefined
```

Functions - More on scope 3

- Forget the var?
  - You get a global!

```
function doSomething () {

  c = 1;

  function doMore () {

    // c = 1
  }
}
// c = 1
```

Functions
- Closures (1/2)

- Functions can bind scope
  - Even after their outer function have returned!

```
var counter = function () {
  var i = 0;

  return function () {
    return i++;
  }
};

var inc = counter();
inc(); // 0
inc(); // 1
```

"This is a really good part"

Functions
- Closures (2/2)

06 Globally ugly, privately slow, closingly sweet

- Functions can run immediately!

```
var inc = function () {
  var i = 0;

  return function () {
    return i++;
  }
}();

inc(); // 0
inc(); // 1
```

Functions - Context

- Functions run in different context
  - Depending on how they are used
- Can mainly run in 4 ways
  - Function invocation
  - Constructor invocation
  - Method invocation
  - Apply/Call invocation

Functions

07 What is this

- Function invocation

```
var countArgs = function () {

  // this === window

  return arguments.length;
};

countArgs(); // 0
```

Functions
08 Constructing objects
- Constructor invocation 1

- Functions used with the new keyword

```
function Person (name) {

  // this === each instance

  this.name = name;
}

var bob = new Person('bob');
var ed = new Person('ed');
```

Functions
- Constructor invocation 2

- But there's a gotcha!

```
function Person (name) {
  this.name = name;
}

var bob = Person('bob');
var ed = new Person('ed');
```

window.name === 'bob' // true

wtf

Functions
- Method invocation


```
var o = {
  retur : 1,
  method : function () {

    // this === o

    return this.retur;
  }
};

o.method() // 1
```

Functions
- Apply/Call invocation


```
function doStuff() { return this.speed; }


var obj = { speed : 2 };


doStuff.apply(obj, [arg1, arg2]);

doStuff.call(obj, arg1, arg2);
```

Inheritance

- Pseudoclassical Inheritance
- Prototypal/Differential Inheritance
- Functional Inheritance

Objects - Prototype

- All objects inherit from their prototype
- Inherit directly from objects
  - Properties
  - Methods
- Bound upon object creation

```
var first = {};
first.toString(); // [object Object]
```

Objects - Prototype

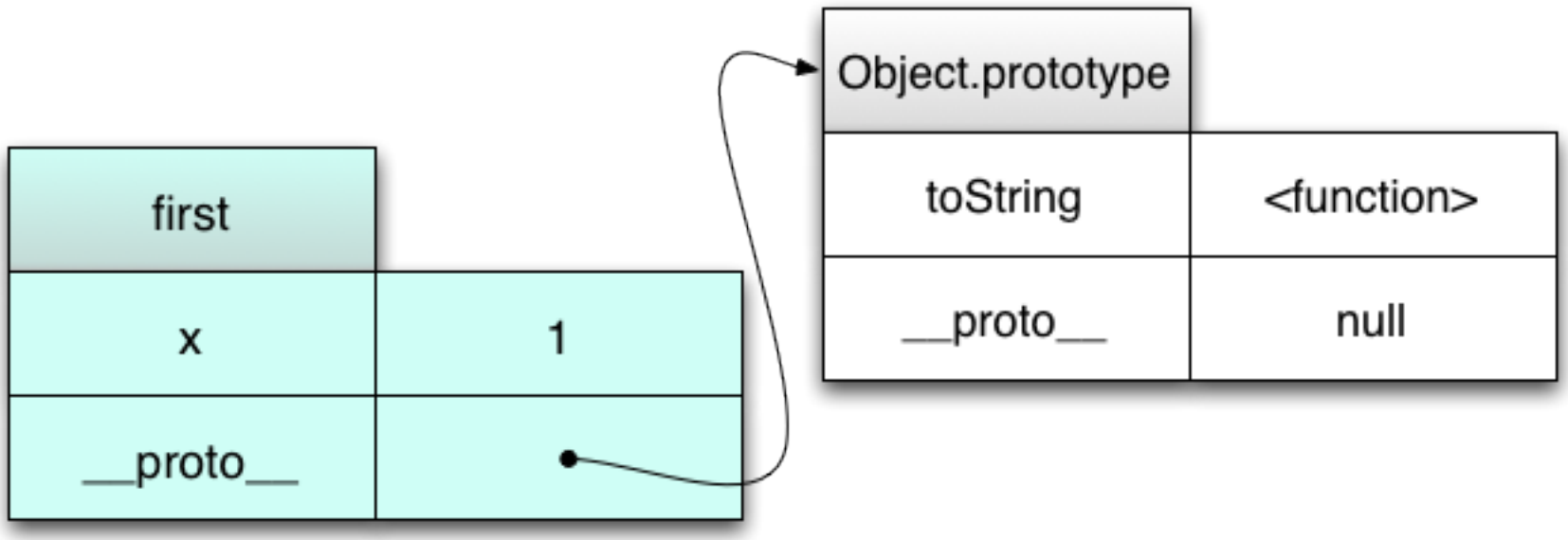
```
function Object () {}

Object.prototype = {
  toString: function () {
    return '[object Object]';
  }
};

var first = new Object();
first.toString(); // [object Object]
```

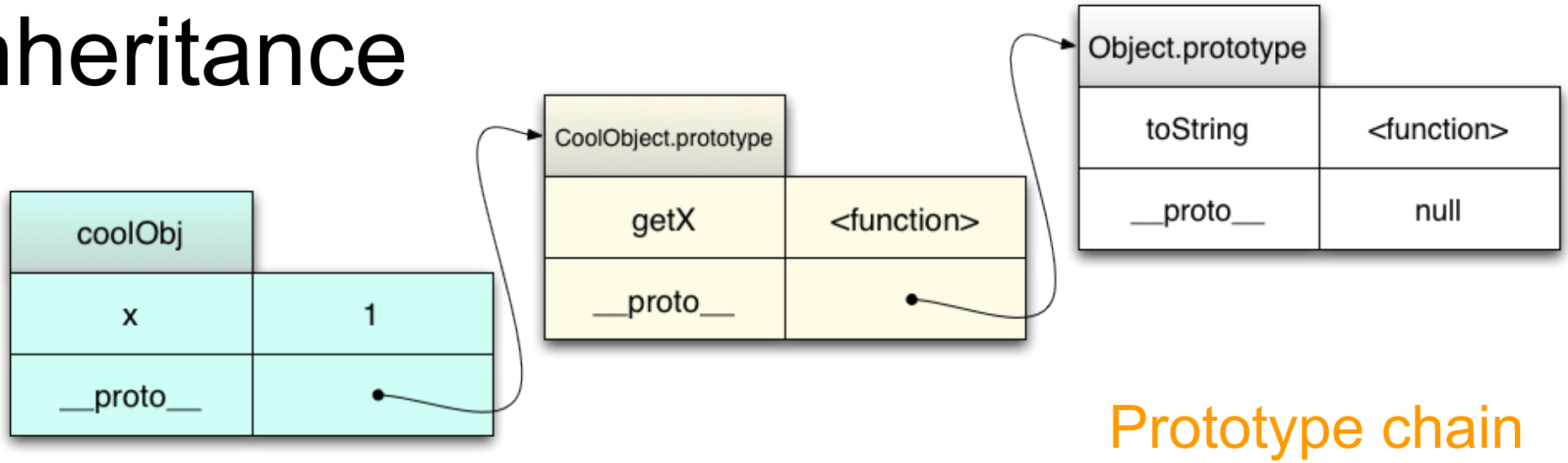
Objects - Prototype

```
var first = new Object();
first.x = 1;
// first.__proto__ = Object.prototype;
```



dmitrysoshnikov.com/ecmascript/javascript-the-core/

Pseudoclassical inheritance



Prototype chain

```
function CoolObject (x) {
  this.x = x;
}
CoolObject.prototype = {
  getX : function () {
    return this.x;
  }
};
var coolObj = new CoolObject(1);
```

Pseudoclassical inheritance

- Intended to look object oriented - looks quite alien?

- Remember to use new!

```
new Car();
```

  - Clobbering global variables
  - No compile warning, No runtime warning
  - Convention
    - All constructor functions start with an initial capital

" Comfort to unexperienced JavaScript programmers "
" Induce unnecessary complex hierarchies "
" ..motivated by constraints of static type checking "

=> JavaScript has more and better options

Prototypal/Differential Inheritance

- Dispence with classes, make useful objects
- Specify differences

```
function create (proto) {
  function F () {};
  F.prototype = proto;
  return new F();
}
var utilities = {
  uberMethod: function () {};
};
var o = create(utilities);
o.uberMethod();
```

Functional Inheritance


```
var person = function (spec, shared) {
  var that = {};

  // shared.persons++

  that.getName = function () {
    return spec.name;
  };

  return that;
};
var bob = person({ name: 'bob'});
var ed = person({ name: 'ed'});
```

Module pattern

```
var lib = {};
lib.module = (function () {

  var privateVariable;

  var privateFunction = function () {};

  return {
    publicProperty: 1,
    privilegedMethod: function (arg) {
      // privateVariable = arg;
    }
  };
})();
```

Bad Parts - Arrays

```
var arr = [ 1, 2, 3 ];
arr[999] = 0;
arr.length  // 1000
```

WTF

```
delete arr[1];
arr // [1, undefined, 3]
```

Bad Parts - For in

- For in loops properties from the prototype chain

```
var key;
for (key in o) {
  if (o.hasOwnProperty(key)) {

  }
}
```

Bad Parts - Eval is evil

- Don't do

```
eval('var value = obj.' + key);
```

- Instead, do

```
var value = obj[key];
```

- Don't do

```
var f = new Function("alert('hei');");
setTimeout("alert('hei')", 1000);
setInterval("alert('hei')", 1000);
```

- Instead, do

```
var f = function () { alert('hei'); };
setTimeout(f, 1000);
```

Bad Parts - Falsy values

- false
- null
- undefined
- The empty string '' or ""
- The number 0
- The number NaN

- && and ||
  - Returns the actual value of the expressions that stops the evaluation

```
function Car (name) {
  this.name = name || 'default name';
}
```

Bad Parts - Transitivity
- Come again?

```
0 == '0'           // true
0 == ''            // true
'' == '0'          // false

false == 'false'   // false
false == '0'       // true
undefined == false // false
undefined == true  // false
false == null      // false
null == undefined  // true

' \t\r\n ' == 0    // true
```

WTF

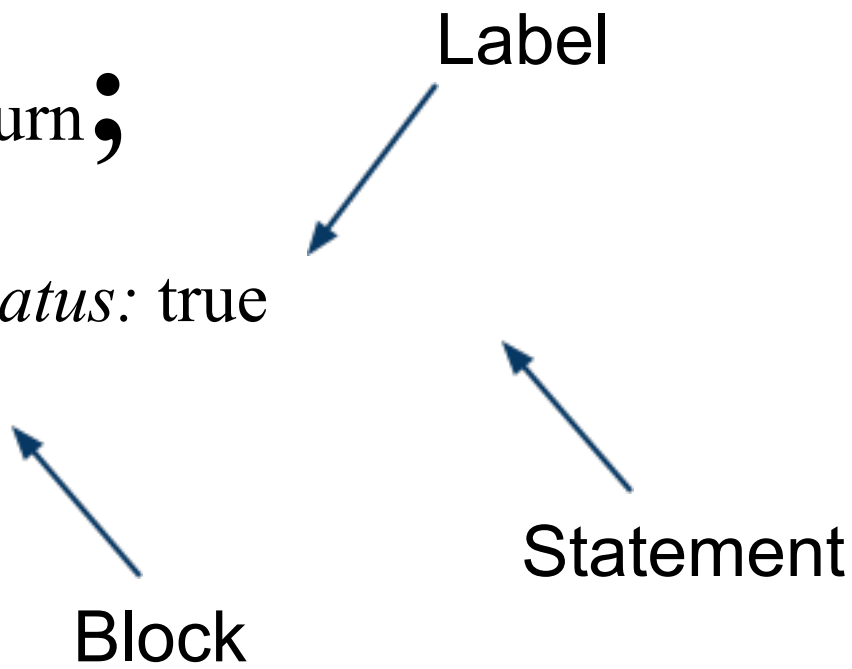
Bad Parts - Semicolon insertion

```
function getStatus () {

  return
  {
    status: true
  };
}
```

Bad Parts - semicolon insertion

```
function getStatus () {
  return;
  {
    status: true
  };
}
```



LOL

Bad Parts - arguments is not an array

- Altough, it pretends to be

```
{
  '0': 1,
  '1': 2
}
```

- You can fix it..

```
function f () {
  var slice = Array.prototype.slice;
  return slice.apply(arguments);
};
f(1, 2); // [ 1, 2 ]
```

Bad Parts - keywords

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

undefined, NaN, Infinity

Bad Parts - void

- *void* is an operator
- takes an operand
- returns undefined

" This is not useful, avoid void "

```
void 1 // undefined
```