



# A (Re)-Introduction to JavaScript

Simon Willison

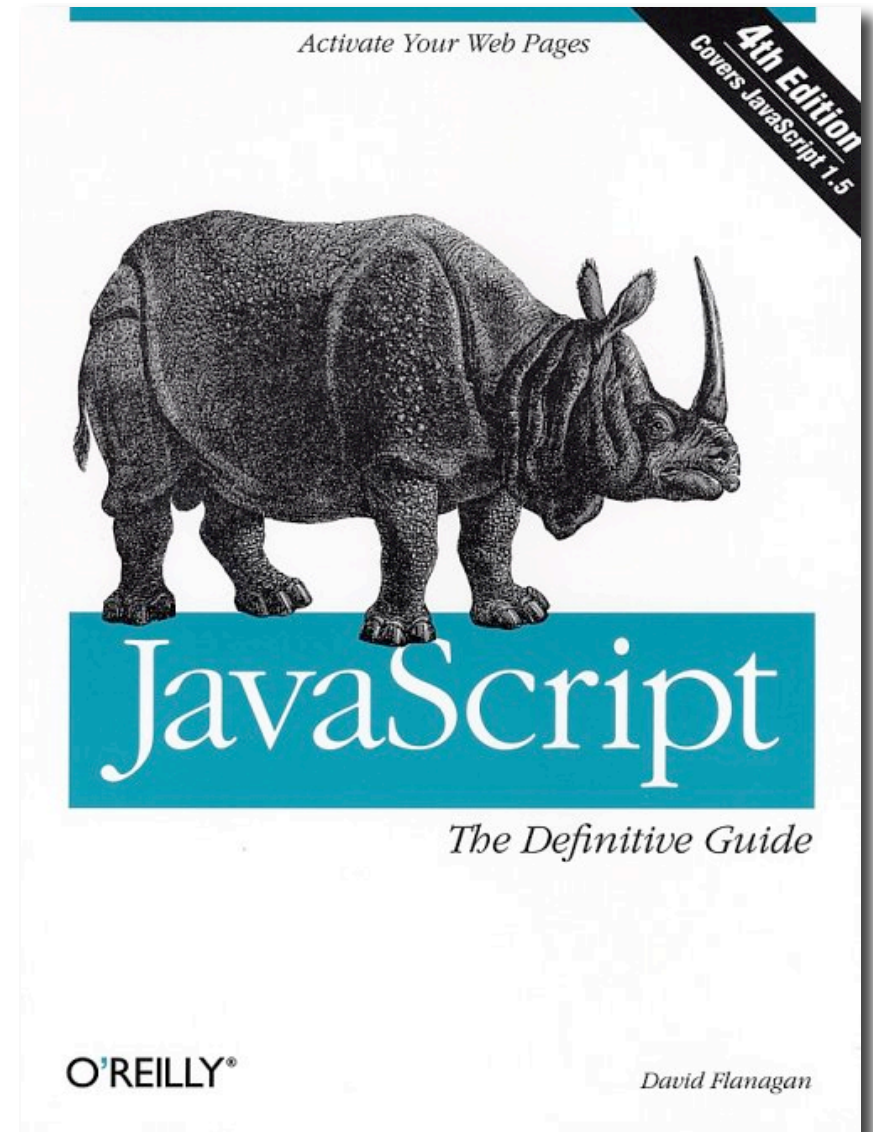
ETech, March 6th, 2005

# Coming up...

- Numbers, Strings and things
- Variables and Operators
- Control Structures
- Objects and Arrays
- Functions
- Object constructors and inheritance
- Inner functions and closures
- Memory Leaks... and more

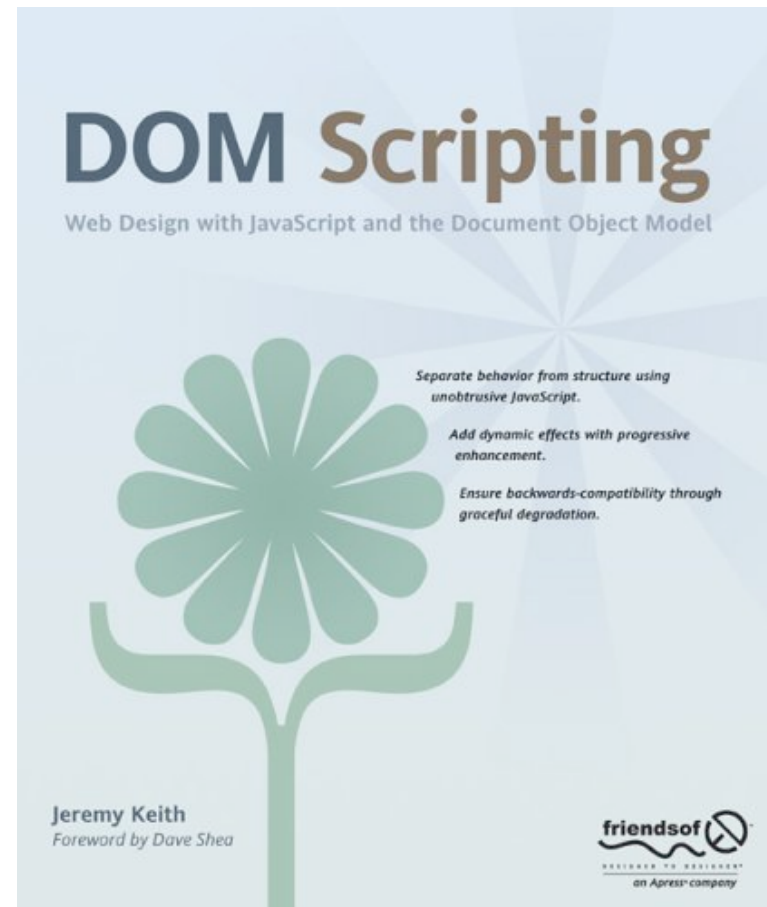
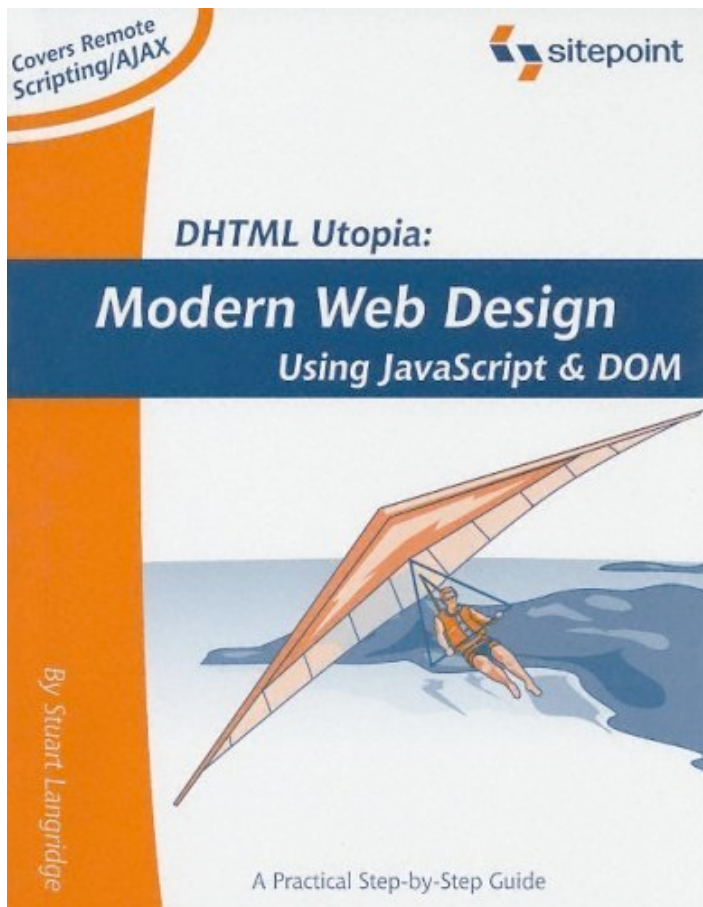
# Book recommendation

- **JavaScript: The Definitive Guide**
  - by David Flanagan
- Comprehensive overview of the language in the first 20 chapters
- New edition out later this year



# More book recommendations

- For practical applications of this stuff:



# You will need...

- Firefox
  - Any web browser can run JavaScript, but Firefox has the best tools for developers
- Jesse Ruderman's 'shell' tool
  - <http://www.squarefree.com/shell/>
- A text editor may come in handy too



# In the words of its creator

*JavaScript was a rushed little hack for Netscape 2 that was then frozen prematurely during the browser wars, and evolved significantly only once by ECMA. So its early flaws were never fixed, and worse, no virtuous cycle of fine-grained community feedback in the form of standard library code construction, and revision of the foundational parts of the language to better support the standard library and its common use-cases, ever occurred.*

# The History of JavaScript

*(in 30 seconds)*

# JavaScript was...

*Invented by Brendan Eich in 1995 for Netscape; originally called LiveScript, current name being an ill-fated marketing decision to tie in with Sun's newly released Java; adapted by Microsoft as JScript for IE 3 in 1996; standardised as ECMAScript in 1997; sort-of included in Flash as ActionScript; updated to ECMAScript 3rd edition in 1998*

# Not quite general purpose

- No direct concept of input or output
- Runs within a host environment
  - Web browser
  - Adobe Acrobat
  - Photoshop
  - Windows Scripting Host
  - Yahoo! Widget Engine
  - and more...

# SYNTAX HACKS

*Tips & Tools for Better  
Writing and Editing*



## Syntax

O'REILLY®

*Brian Sawyer*

# Reserved words

break else new var  
case finally return void  
catch for switch while  
continue function this with  
default if throw  
delete in try  
do instanceof typeof

abstract enum int short  
boolean export interface static  
byte extends long super  
char final native synchronized  
class float package throws  
const goto private transient  
debugger implements protected volatile  
double import public

# Style recommendations

- Avoid semi-colon insertion like the plague

■     ■     ■  
;     ;     ; ← **friends**

- Declare variables religiously with 'var'
- Global variables are evil. Avoid them if you can.
- Indent consistently



Types

# JavaScript types

- Numbers
- Strings
- Booleans
- Functions
- Objects

# JavaScript types (improved)

- Number
- String
- Boolean
- Object
  - Function
  - Array
  - Date
  - RegExp
- Null
- Undefined



# Numbers

# Numbers

- "double-precision 64-bit format IEEE 754 values"
- No such thing as an integer
$$0.1 + 0.2 = 0.300000000000000000000004$$
$$3 + 5.3$$
$$28 \% 6 \text{ etc...}$$
- Math namespace for advanced operations

# Math stuff

- `Math.PI`, `Math.E`, `Math.LN10`,  
`Math.LN2`, `Math.LOG10E`,  
`Math.SQRT1_2`, `Math.SQRT2`
- **`Math.abs(x)`**, `Math.acos(x)`,  
`Math.asin(x)`, `Math.atan(x)`,  
`Math.atan2(y, x)`, **`Math.ceil(x)`**,  
`Math.cos(x)`, `Math.exp(x)`,  
**`Math.floor(x)`**, `Math.log(x)`,  
**`Math.max(x, ..)`**, **`Math.min(x, ..)`**,  
`Math.pow(x, y)`, **`Math.random()`**,  
**`Math.round(x)`**, `Math.sin(x)`,  
`Math.sqrt(x)`, `Math.tan(x)`

# parseInt (and parseFloat)

- parseInt converts a string to a number

```
> parseInt( "123" );
```

```
123
```

```
> parseInt( "010" );
```

```
8
```

Always specify the base

- !?

```
> parseInt( "010", 10 );
```

```
10
```

```
> parseInt( "11", 2)
```

```
3
```

# NaN and Infinity

```
> parseInt("hello", 10)
```

```
NaN
```

- NaN is toxic

```
> NaN + 5
```

```
NaN
```

```
> isNaN(NaN)
```

```
true
```

- Special values for Infinity and -Infinity:

```
> 1 / 0
```

```
Infinity
```

```
> -1 / 0
```

```
-Infinity
```

# Strings

# Strings

- Sequences of characters
- Sequences of *unicode* characters (16 bit)

```
"\u0020"
```

- A character is a string of length 1

```
> "hello".length
```

```
5
```

- Strings are objects!

# String methods

```
> "hello".charAt(0)
```

```
h
```

```
> "hello, world".replace("hello",  
"goodbye")
```

```
goodbye, world
```

```
> "hello".toUpperCase()
```

```
HELLO
```

# More string methods

`s.charAt(pos)` `s.charCodeAt(pos)`

`s.concat(s1, ..)`

`s.indexOf(s1, start)`

`s.lastIndexOf(s1, startPos)`

`s.localeCompare(s1)` `s.match(re)`

`s.replace(search, replace)`

`s.search(re)` `s.slice(start, end)`

`s.split(separator, limit)`

`s.substring(start, end)`

`s.toLowerCase()` `s.toLocaleLowerCase()`

`s.toUpperCase()` `s.toLocaleUpperCase()`

# Null and undefined

- null = deliberately no value
- undefined = no value assigned yet
  - Variables declared but not initialised
  - Object/array members that don't exist
  - (More on this later)

# Booleans

- Boolean type: true or false
- Everything else is "truthy" or "falsy"
- 0, "", NaN, null, undefined are falsy
- Everything else is truthy
- Boolean operations: &&, || and !
- Convert any value to its boolean equivalent by applying not twice:

```
> !! ""
```

```
false
```

```
> !! 234
```

```
true
```



# Variables and operators

# Variable declaration

- New variables in JavaScript are declared using the `var` keyword:

```
var a;
```

```
var name = "simon";
```

- If you declare a variable without assigning it to anything, its value is undefined.

If you forget the `var`, you get a global variable. **Never, ever do this** - not even if you mean it.

# Operators

Numeric operators: +, −, \*, / and %

- Compound assignment operators:

`+=, -=, *=, /=, %=`

- Increment and decrement:

`a++, ++a, b--, --b`

- String concatenation:

`> "hello" + " world"`

`hello world`

# Type coercion

```
> "3" + 4 + 5
```

```
345
```

```
> 3 + 4 + "5"
```

```
75
```

- Adding an empty string to something else converts it to a string.

# Comparison

- `>`, `<`, `>=`, `<=` work for numbers and strings
- Equality tests use `==` and `!=` ... sort of
  - `> "dog" == "dog"`  
`true`
  - `> 1 == true`  
`true`
- `===` and `!==` avoid type coercion
  - `> 1 === true`  
`false`
  - `> true === true`  
`true`

# The typeof operator

typeof v

number	'number'
string	'string'
boolean	'boolean'
function	'function'
object	'object'
array	<b>'object'</b>
null	<b>'object'</b>
undefined	'undefined'

# Control structures

# if statements

```
var name = "kittens";  
if (name == "puppies") {  
    name += "!";  
} else if (name == "kittens") {  
    name += "!!";  
} else {  
    name = "!" + name;  
}  
name == "kittens!!"
```



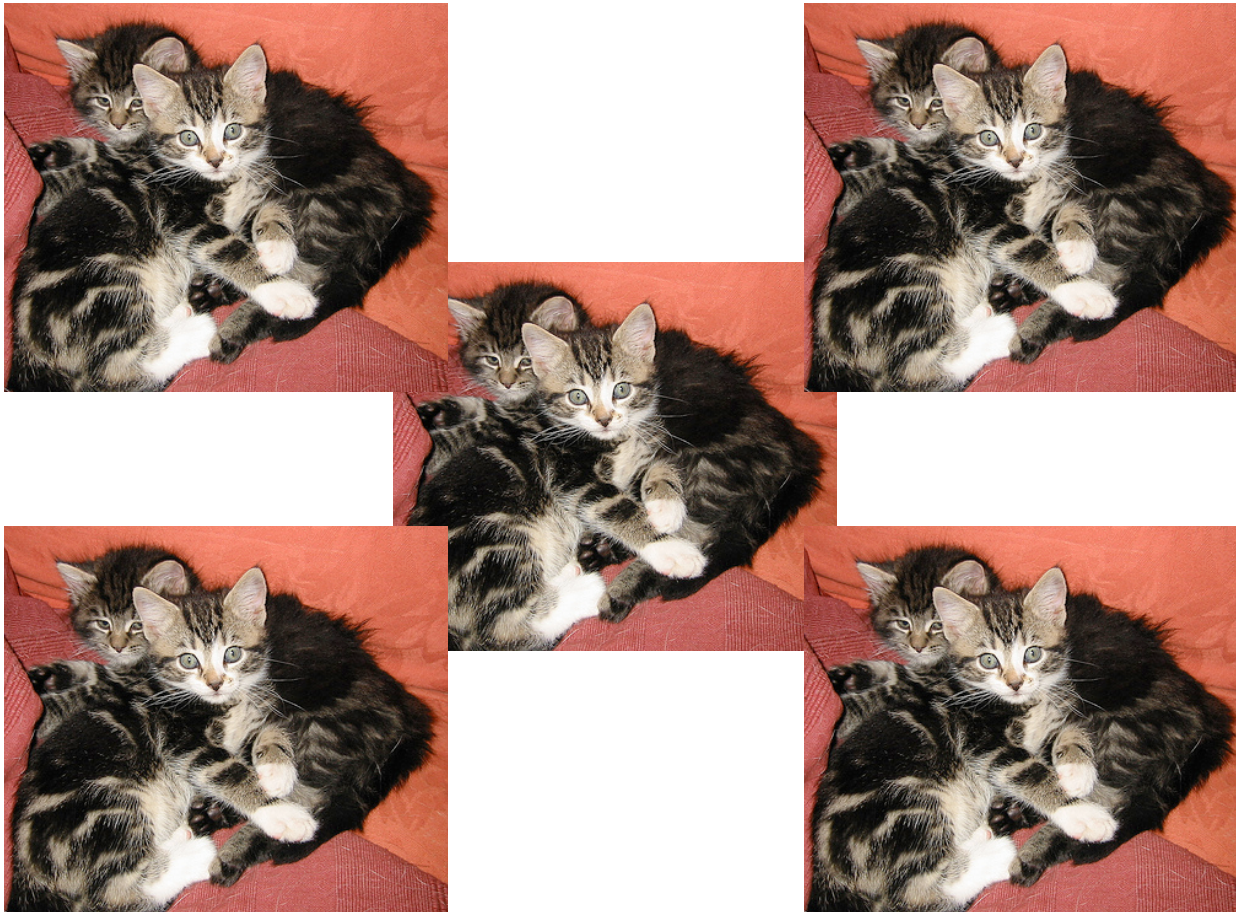
# while and do-while

```
while (true) {  
    // an infinite loop!  
}
```

```
do {  
    var input = get_input();  
} while (inputIsValid(input))
```

# for loops

```
for (var i = 0; i < 5; i++) {  
    // Will execute 5 times  
}
```



# switch statement

- Multiple branches depending on a number or string

```
switch(action) {  
    case 'draw':  
        drawit();  
        break;  
    case 'eat':  
        eatit();  
        break;  
    default:  
        donothing();  
}
```

# fall through

```
switch(a) {  
    case 1: // fallthrough  
    case 2:  
        eatit();  
        break;  
    default:  
        donothing();  
}
```

- Deliberate labelling of fall through is good practise

# Switch expressions

- Expressions are allowed
- Comparisons take place using `===`

```
switch(1 + 3):  
    case 2 + 2:  
        yay();  
        break;  
    default:  
        neverhappens();  
}
```

# Short-circuit logic

- The `&&` and `||` operators use short-circuit logic: they will execute their second operand dependant on the first.
- This is useful for checking for null objects before accessing their attributes:

```
var name = o && o.getName();
```

- Or for setting default values:

```
var name = otherName || "default";
```

# The tertiary operator

- One-line conditional statements

```
var ok = (age > 18) ? "yes" : "no";
```

- Easy to abuse; use with caution

# Exceptions

```
try {  
    // Statements in which  
    // exceptions might be thrown  
} catch(error) {  
    // Statements that execute  
    // in the event of an exception  
} finally {  
    // Statements that execute  
    // afterward either way  
}
```

```
throw new Error("An error!");  
throw "Another error!";
```



# Objects



# Objects

- Simple name-value pairs, as seen in:
  - Dictionaries in Python
  - Hashes in Perl and Ruby
  - Hash tables in C and C++
  - HashMaps in Java
  - Associative arrays in PHP
- Very common, versatile data structure
- Name part is a string; value can be anything

# Basic object creation

```
var obj = new Object();
```

- Or:

```
var obj = {};
```

- These are semantically equivalent; the second is called object literal syntax and is more convenient.

# Property access

```
obj.name = "Simon"
```

```
var name = obj.name;
```

- Or...

```
obj["name"] = "Simon";
```

```
var name = obj["name"];
```

- Semantically equivalent; the second uses strings so can be decided at run-time (and can be used for reserved words)

# Object literal syntax

```
var obj = {  
  name: "Carrot",  
  "for": "Max",  
  details: {  
    color: "orange",  
    size: 12  
  }  
}
```

```
> obj.details.color  
orange
```

```
> obj["details"]["size"]  
12
```

# for (var attr in obj)

- You can iterate over the keys of an object:

```
var obj = { 'name': 'Simon', 'age': 25};  
for (var attr in obj) {  
    print (attr + ' = ' + obj[attr]);  
}
```

- Don't attempt this with arrays (coming up next). There are safer ways of iterating over them.



# Arrays

# Arrays

- A special type of object: Keys are whole numbers, not strings.
- Use [] syntax, just like objects

```
> var a = new Array( );
```

```
> a[0] = "dog";
```

```
> a[1] = "cat";
```

```
> a[2] = "hen";
```

```
> a.length
```

```
3
```

# Array literals

- More convenient notation:

```
> var a = [ "dog", "cat", "hen" ];
```

```
> a.length
```

```
3
```

# array.length

```
> var a = [ "dog", "cat", "hen" ];
```

```
> a[100] = "fox";
```

```
> a.length
```

```
101
```

```
typeof a[90] == 'undefined'
```

- array.length is always one more than the highest index
- The safest way to append new items is:

```
a[a.length] = item;
```

# Array iteration

```
for (var i = 0; i < a.length; i++) {  
    // Do something with a[i]  
}
```

- Improve this by caching a.length at start:

```
for (var i = 0, j = a.length; i < j; i++) {  
    // Do something with a[i]  
}
```

# Even better iteration

```
for (var i = 0, item; item = a[i]; i++) {  
    // Do something with item  
}
```

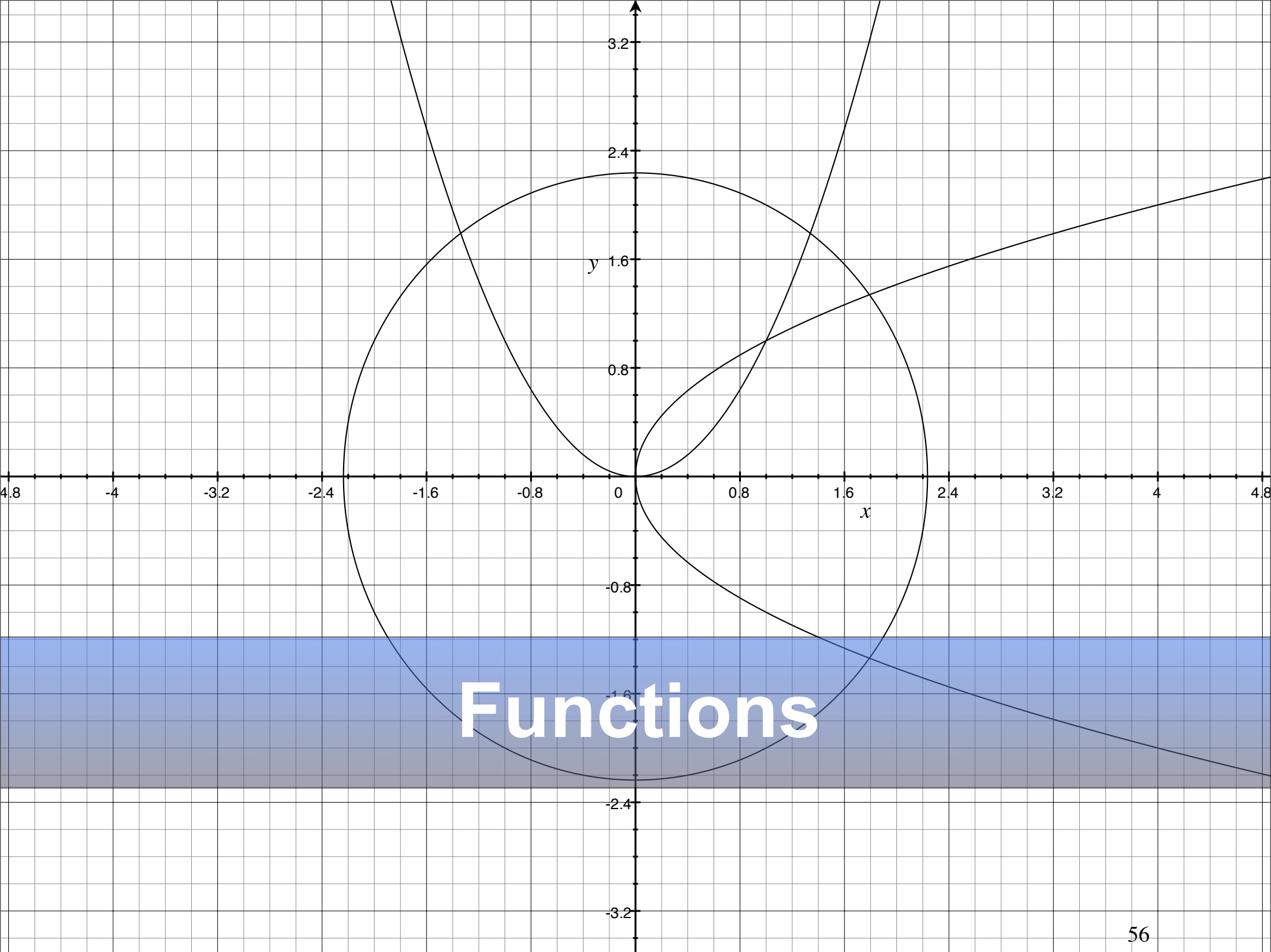
- This trick only works with arrays that are known not to contain falsy values. The following array will break with the above idiom:

```
var a = [10, "dog", false, "elephant"];
```

- (Yes, you can have mixed content in arrays)

# Array methods

```
a.toString(), a.toLocaleString(),  
a.concat(item, ..), a.join(sep), a.pop(),  
a.push(item, ..), a.reverse(), a.shift(),  
a.slice(start, end), a.sort(cmpfn),  
a.splice(start, delcount, [item]..),  
a.unshift([item]..)
```



# Functions

- Don't get much simpler than this:

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

- If nothing is explicitly returned, return value is undefined

# Arguments

- Parameters: *"They're more like... guidelines"*
- Missing parameters are treated as undefined:

```
> add( )
```

```
Nan // addition on undefined
```

- You can pass in more arguments than expected:

```
> add(2, 3, 4)
```

```
5 // added the first two; 4 was ignored
```

- How is this behaviour useful?

# arguments

- The arguments special variable provides access to arguments as an array-like object

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

```
> add(2, 3, 4, 5)  
14
```

# avg()

- More useful: an averaging function:

```
function avg() {  
    var sum = 0;  
    for (var i = 0, j = arguments.length;  
        i < j; i++) {  
        sum += arguments[i];  
    }  
    return sum / arguments.length;  
}
```

# Averaging an array?

- Our fancy multi-argument function isn't much good if our data is already in an array. Do we have to rewrite it?

```
function avgArray(arr) {  
    var sum = 0;  
    for (var i = 0, j = arr.length;  
        i < j; i++) {  
        sum += arguments[i];  
    }  
    return sum / arr.length;  
}
```

# Using avg() with an array

- That's not necessary:

```
> avg.apply(null, [2, 3, 4, 5])
```

```
3.5
```

- Functions are objects with methods too!
- The apply method takes an array of arguments as the second argument...
- We'll find out about the first argument a little later

# Anonymous functions

- The following is semantically equivalent to our earlier `avg()` function:

```
var avg = function() {  
    var sum = 0;  
    for (var i = 0, j = arguments.length;  
        i < j; i++) {  
        sum += arguments[i];  
    }  
    return sum / arguments.length;  
}
```

# The block scope trick

- Block scope is a feature of C where every set of braces defines a new scope. It can be simulated in JavaScript:

```
var a = 1;
var b = 2;
(function() {
    var b = 3;
    a += b;
})();
> a
4
> b
2
```

# Recursive functions

```
function countChars(elm) {  
    if (elm.nodeType == 3) { // TEXT_NODE  
        return elm.nodeValue.length;  
    }  
    var count = 0;  
    for (var i = 0, child;  
        child = elm.childNodes[i]; i++) {  
        count += countChars(child);  
    }  
    return count;  
}
```

# arguments.callee

- arguments.callee is the current function:

```
var charsInBody = (function(elm) {  
    if (elm.nodeType == 3) { // TEXT_NODE  
        return elm.nodeValue.length;  
    }  
    var count = 0;  
    for (var i = 0, child;  
        child = elm.childNodes[i]; i++) {  
        count += arguments.callee(child);  
    }  
    return count;  
})(document.body);
```

# arguments.callee to save state

- This function remembers how many times it has been called:

```
function counter() {  
    if (!arguments.callee.count) {  
        arguments.callee.count = 0;  
    }  
    return arguments.callee.count++;  
}  
> counter()  
0  
> counter()  
1
```



# Constructors

# Functions and objects

```
function makePerson(first, last) {  
    return {  
        first: first,  
        last: last  
    }  
}  
  
function personFullName(person) {  
    return person.first + ' ' +  
        person.last;  
}  
  
function personFullNameReversed(person) {  
    return person.last + ', ' +  
        person.first  
}
```

# Functions and objects (II)

```
> s = makePerson("Simon",  
"Willison");
```

```
> personFullName(s)
```

```
Simon Willison
```

```
> personFullNameReversed(s)
```

```
Willison, Simon
```

- Surely we can attach functions to the objects themselves?

# Methods, first try

```
function makePerson(first, last) {  
  return {  
    first: first,  
    last: last,  
    fullName: function() {  
      return this.first + ' ' +  
        this.last;  
    },  
    fullNameReversed: function() {  
      return this.last + ', ' +  
        this.first;  
    }  
  }  
}
```

# Using methods

```
> s = makePerson("Simon",  
"Willison")
```

```
> s.fullName()
```

```
Simon Willison
```

```
> s.fullNameReversed()
```

```
Willison, Simon
```

# dot notation is required

```
> s = makePerson( "Simon",  
  "Willison" )  
  
> var fullName = s.fullName;  
  
> fullName()  
  
undefined undefined
```

- If you call a function without using dot notation, 'this' is set to the global object. In the browser, this is the same as 'window'.

# Constructors

```
function Person(first, last) {  
    this.first = first;  
    this.last = last;  
    this.fullName = function() {  
        return this.first + ' ' +  
            this.last;  
    }  
    this.fullNameReversed = function() {  
        return this.last + ', ' +  
            this.first;  
    }  
}  
  
var s = new Person("Simon", "Willison");
```

# new

- 'new' creates a new empty object and calls the specified function with 'this' set to that object.
- These constructor functions should be Capitalised, as a reminder that they are designed to be called using 'new'
- This is key to understanding JavaScript's object model

# Sharing methods

```
function personFullName() {  
    return this.first + ' ' +  
        this.last;  
}  
function personFullNameReversed() {  
    return this.last + ', ' +  
        this.first;  
}  
function Person(first, last) {  
    this.first = first;  
    this.last = last;  
    this.fullName = personFullName;  
    this.fullNameReversed =  
        personFullNameReversed;  
}
```

# Person.prototype

```
function Person(first, last) {  
    this.first = first;  
    this.last = last;  
}  
Person.prototype.fullName =  
    function() {  
        return this.first + ' ' +  
            this.last;  
    }  
Person.prototype.fullNameReversed =  
    function() {  
        return this.last + ', ' +  
            this.first;  
    }  
}
```

# instanceof

- The instanceof operator can be used to test the type of an object, based on its constructor (and prototype hierarchy, explained in the next section).

```
var a = [1, 2, 3];  
a instanceof Array  
true  
a instanceof Object  
true  
a instanceof String  
false
```



# Inheritance

# Adding to an existing set of objects

```
> s = new Person("Simon",  
"Willison");  
> s.firstNameCaps();  
TypeError on line 1:  
s.firstNameCaps is not a function  
> Person.prototype.firstNameCaps =  
function() {  
    return this.first.toUpperCase()  
}  
> s.firstNameCaps()  
SIMON
```

# Extending core objects

```
> var s = "Simon";
```

```
> s.reversed()
```

**TypeError: s.reversed is not a function**

```
> String.prototype.reversed = function() {  
  var r = '';  
  for (var i = this.length - 1; i >= 0; i--){  
    r += this[i];  
  }  
  return r;  
}
```

```
> s.reversed()
```

nomiS

```
> "This can now be reversed".reversed()
```

desrever eb won nac sihT

# Object.prototype

- All prototype chains terminate at Object.prototype
- Its methods include toString(), which we can over-ride:

```
> var s = new Person("Simon", "Willison");  
> s  
[object Object]  
> Person.prototype.toString = function() {  
    return '<Person: ' + this.fullName() + '>';  
}  
> s  
<Person: Simon Willison>
```

# Perils of modifying `Object.prototype`

- Remember `for (var attr in obj)?`
- It will include stuff that's been newly added to `Object.prototype`
- This stupid behaviour is sadly baked in to the language
- Some libraries (Prototype is a prime offender) do it anyway

# So what about inheritance?

- Problem: JavaScript is a prototype-based language, but it pretends to be a class-based language
- As a result, it doesn't do either very well
- Inheritance doesn't quite behave how you would expect

# Here's a special kind of person

```
function Geek() {  
    Person.apply(this, arguments);  
    this.geekLevel = 5;  
}  
Geek.prototype = new Person();  
Geek.prototype.setLevel = function(lvl) {  
    this.geekLevel = lvl;  
}  
Geek.prototype.getLevel = function() {  
    return this.geekLevel;  
}  
> s = new Geek("Simon", "Willison")  
> s.fullName()  
Simon Willison  
> s.getLevel()  
5
```

# new Person()?

- We're using an *instance* of the Person object as our prototype
- We have to do this, because we need to be able to modify our prototype to add new methods that are only available to Geek instances
- This is counter-intuitive and, well, a bit dumb



# Solutions?

- Design classes with inheritance in mind - don't do anything important in the constructor (that might break if you create an empty instance for use as a prototype)
- Use one of the many workarounds
  - Prototype's `Class.create()`
  - The stuff you get by searching for "javascript inheritance" on the Web

# Higher order functions

# Functions are first-class objects

- In English...
  - A function is just another object
  - You can store it in a variable
  - You can pass it to another function
  - You can return it *from* a function

# VAT

**VAT is England's national sales tax - 17.5%**

```
var prices = [10, 8, 9.50];  
var pricesVat = [];  
for (var i = 0; i < prices.length; i++)  
{  
    pricesVat[i] = prices[i] * 1.175;  
}
```

# arrayMap

```
function arrayMap(array, func) {  
    var result = [];  
    for (var i = 0; i < array.length; i++) {  
        result[i] = func(array[i]);  
    }  
    return result;  
}
```

```
function calcVat(price) {  
    return price * 1.175;  
}
```

```
var prices = [10, 8, 9.50];  
pricesVat = arrayMap(prices, calcVat);
```

# salesTaxFactory

**What if we want to calculate sales tax at 4%?**

```
function salesTaxFactory(percent) {  
  function func(price) {  
    return price + (percent / 100) * price;  
  }  
  return func;  
}
```

```
calcVat = salesTaxFactory(17.5);  
calc4 = salesTaxFactory(4);
```

```
pricesVat = arrayMap(prices, calcVat);  
prices4 = arrayMay(prices, calc4);
```

# An operation factory

```
function makeOp(op, y) {  
  switch (op) {  
    case '+':  
      return function(x) { return x + y };  
    case '-':  
      return function(x) { return x - y };  
    case '/':  
      return function(x) { return x / y };  
    case '*':  
      return function(x) { return x * y };  
    default:  
      return function(x) { return x };  
  }  
}  
  
var third = makeOp('/', 3);  
var dbl = makeOp('*', 2);  
print(third(24));  
print(dbl(5));
```

# Closures

- The previous code was an example of *closures* in action
- A closure is a function that has captured the scope in which it was defined
- Actually, functions in JavaScript have a *scope chain* (similar to the prototype chain)

# What does this do?

```
function openLinksInNewWindows() {  
    for (var i = 0; i < document.links.length; i++) {  
        document.links[i].onclick = function() {  
            window.open(document.links[i].href);  
            return false;  
        }  
    }  
}
```

- The onclick function is a closure which refers back to the original scope; it does NOT retain a copy of the *i* variable at the time the function was created. By the time the onclick function is executed, *i* will be the last value assigned by the loop.



**Singleton**

# Namespace pollution

- JavaScript shares a single global namespace
- It's easy to clobber other people's functions and variables
- It's easy for other people to clobber yours
- The less code affecting the global namespace the better

# The singleton pattern

```
var simon = (function() {  
  var myVar = 5; // Private variable  
  function init(x) {  
    // ... can access myVar and doPrivate  
  }  
  function doPrivate(x) {  
    // ... invisible to the outside world  
  }  
  function doSomething(x, y) {  
    // ... can access myVar and doPrivate  
  }  
  return {  
    'init': init, 'doSomething': doSomething  
  }  
})();  
  
simon.init(x);
```

# Singleton benefits

- Singleton lets you wrap up a complex application with dozens of functions up in a single, private namespace - a closure
- This lets you expose only the functions that make up your application's external interface

A black fire hydrant is the central focus, with water spraying from its top and a side outlet. A green fire hose is connected to the side outlet. The background shows a city street with a tall brick building, bare trees, and a blue sky. A sign for the 'EMERGENCY RECEIVING DEPT.' is visible on the right. The scene is captured in a cinematic style with a blue color grade.

# Memory leaks

# Internet Explorer sucks

- To understand memory leaks, you have to understand a bit about garbage collection
- Stuff gets freed up automatically when it is no longer in use (both JavaScript objects and host objects, such as DOM nodes)
- IE uses different garbage collectors for JS and for the DOM, and can't handle circular references between them



# This leaks

```
function leak() {  
    var div = document.getElementById( 'd' );  
    div.obj = {  
        'leak': div  
    }  
}
```

- Call it enough times and IE will crash

# This also leaks

```
function sneakyLeak() {  
    var div = document.getElementById('d');  
    div.onclick = function() {  
        alert("hi!");  
    }  
}
```

- Why? Spot the closure!

# Difficulties

- These things can be very hard to detect - especially in large, complex applications where a cycle might occur over many nodes and JavaScript objects
- One solution:

```
function noLeak() {  
    var div = document.getElementById( 'd' );  
    div.onclick = function() {  
        alert("hi!");  
    }  
    div = null;  
}
```

# More solutions

- Most popular JavaScript libraries have systems for attaching and detaching events
- Many of these can automatically free event references when the page is unloaded
- Use these, but always be aware of the problem



# Performance

# Performance tips

- De-reference complex lookups

```
var s = document.getElementById('d').style;  
s.width = '100%';  
s.color = 'red';  
// ...  
s.display = 'block';
```

- ... especially inside loops

```
var lookup = foo.bar.bav;  
for (var i = 0; i < 1000; i++) {  
    lookup.counter += someCalc();  
}
```



# Libraries

# Suggested libraries

The logo for the Dojo Toolkit, featuring the word "dojō" in a stylized, black, serif font with a macron over the 'o'.

[dojotoolkit.org](http://dojotoolkit.org)



[mochikit.com](http://mochikit.com)



[developer.yahoo.net/yui/](http://developer.yahoo.net/yui/)



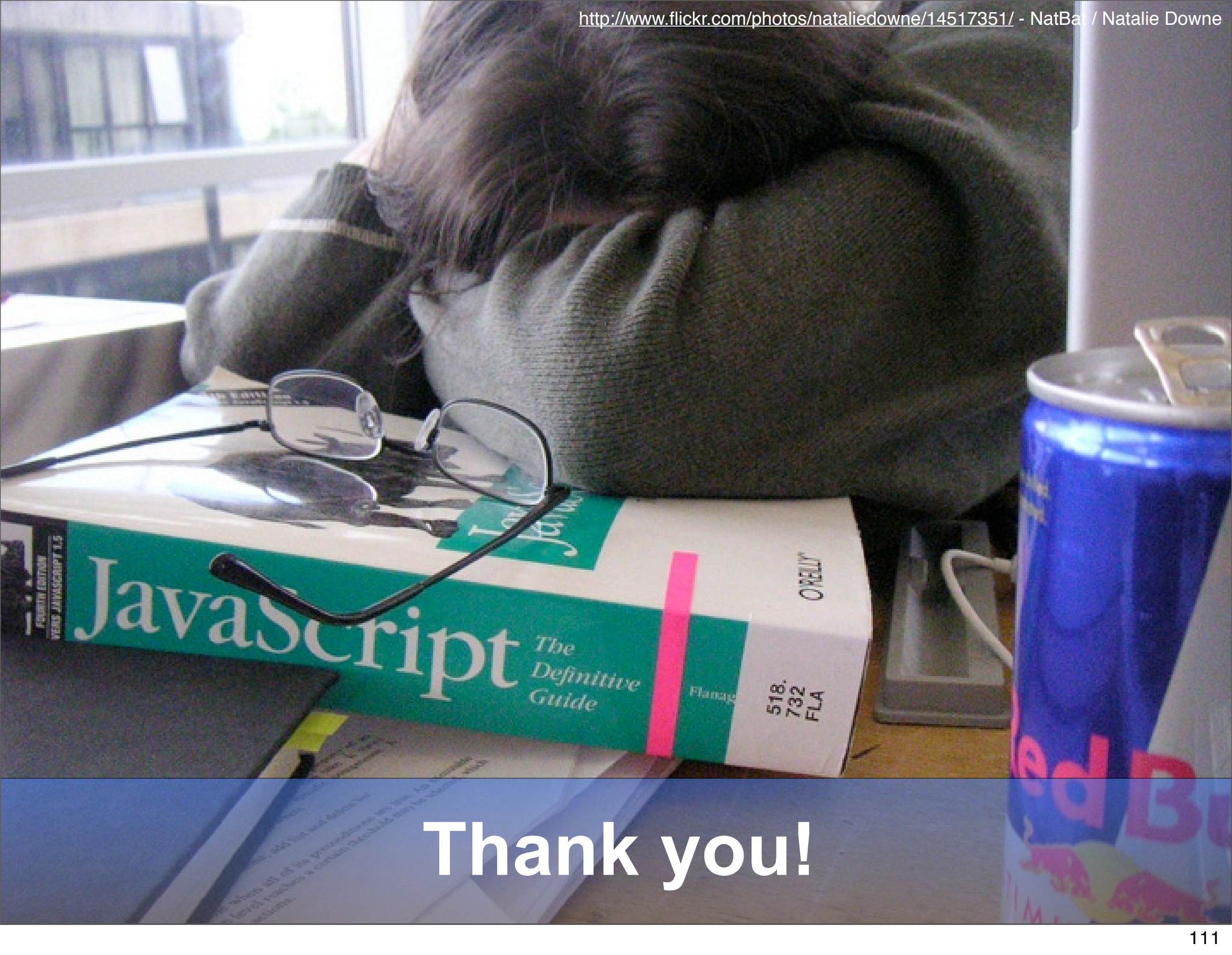
[prototype.conio.net](http://prototype.conio.net)

The script.aculo.us logo, featuring the text "script.aculo.us" in a white, italicized, serif font, set against a solid green rectangular background.

[script.aculo.us](http://script.aculo.us)

# Tell Alex I told you this...

- *Everything in JavaScript is an Object. Even functions*
- *Every object is always mutable*
- *The dot operator is equivalent to de-referencing by hash (e.g., `foo.bar === foo["bar"]`)*
- *The new keyword creates an object that class constructors run inside of, thereby imprinting them*
- *Functions are always closures (combine w/ previous rule to create OOP)*
- *The this keyword is relative to the execution context, not the declaration context*
- *The prototype property is mutable*



Thank you!