JavaScript Workshop

Why This?

- Most people either don’t understand JavaScript, or,
- They think they do but don’t
- Wrongfully given a bad reputation
- Trying raise awareness of it’s potential

Why Me?

- One of the few people that actually like the language
- Responsible for introducing JS framework and development standards in AIB Jaguar Project
- Written & Developing Eclipse plugins to automate compression and code checking of JavaScript files
- Written several jQuery plugins & proof of concepts (Annotations, Constraints, Input Mask)

- Debugging
- Logging
- On the fly CSS/HTML/JS editing
- Profiling
- Monitoring Ajax Requests
- Page Weight Analysis

CSS 101

Overview

- What is CSS
- Cascading Styles
- Selectors
  - Type Selectors
  - Descendant Selector
  - Child Selectors
  - Adjacent Selectors
  - Attribute Selectors
  - Class Selectors
  - ID Selectors
  - Universal Selector
  - Pseudo-class Selectors
- Specificity
- Grouping
- Shorthand Properties
- !important rule

What is CSS?

- CSS – Cascading Style Sheets
- Specify how elements in the DOM should be rendered
- A stylesheet is a collection of CSS Rules which in turn are collections of key/value pair style properties

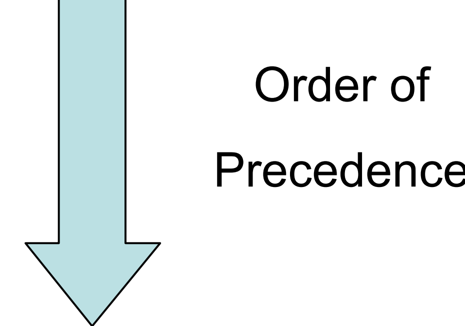
```
body { text-align:right; }
body {
    text-align : right;
    font-size  : 12px;
}
```

- Elements take on the stylistic properties where most specific properties take precedence (hence Cascading)
- Styles are applied progressively
- Styles can be specified in a number of ways

Cascading Styles

- There are 4 ways styles get applied to elements in the DOM. These are:

1. Browser default styles
2. External stylesheet
3. Internal stylesheet
4. Inline styles



Order of

Precedence

- Taking a single page it is possible to progressively enhance it's appearance

Cascading Styles Example

Selectors

```
body { text-align:right; }
```

- A selector is a pattern matching rule that specifies the elements to apply the style rules to.
- These can range from simple type patterns (above) to very complex matching rules.

Type Selectors

- Specify the element type to match i.e. the type of tag to match

```
body { text-align:right; }
```

- Disregards document position entirely

Descendant Selectors

- This is 2 or more selectors separated by spaces where the selector on the right must be a descendant (direct or non-direct) of the one on the left of it

```
ul li { display:inline; }
```

- This selector is saying all li elements who are descendants of ul elements should have the rule applied
- Beware heavily nested trees might cause side effects (e.g. a list of lists where the topmost li should only get styled – this is no applicable to the descendant selector)

Child Selectors

- This is 2 or more selectors separated by “>” where the selector on the right must be a child (or direct descendant) of the one on the left of it

```
ol > li { font-weight:bold; }
```

- Unsupported in Internet Explorer until recently
- Useful for nested elements

Adjacent Selectors

- 2 selectors separated by + where the second selector shares the same parent as the first and immediately follows it.

```
h1 + p { font-size:1.2em; }
```

- Useful for handling padding issues around images, or performing formatting on initial paragraphs etc.
- Again unsupported in Internet Explorer

Attribute Selectors

- Allows matching of elements in the DOM based on their attributes

selector[attribute_name]{...}

  - Matches elements that have the attribute regardless of value

selector[attribute_name=value]{...}

  - Matches elements whose attribute matches the given value

selector[attribute_name~=value]{...}

  - Matches elements whose attribute value (viewed as a space separated list) contains the given value

selector[attribute_name|=value]{...}

  - Matches elements whose attribute value (viewed as a hyphen separated list) first entry matches the value. Primarily for the lang attribute (en-us, en-gb etc.)

Class Selectors

- Matches all elements who have the specified class assigned to them

```
p.italic { font-style:italic; }
```

- HTML specific version of `[class~=value]`

ID Selectors

- Selector matches the element with a specific id

```
#id { font-style:italic; }
```

- Multiple elements with same id can cause unpredictable results (shouldn't happen)

Universal Selector

- Universal Selector matches all elements in the DOM

```
* { font-style:italic; }
```

- Can be ignored in class selectors

```
 .italic { font-style:italic; }
*.italic { font-style:italic; }
```

Pseudo-selectors

- Many different pseudo-selectors

  `:first-child :last-child :link :visited :active :hover :focus :not :first-line :first-letter :nth-child`

- Support is limited

Specificity

- Specificity determines what style rules take precedence
- Each type of selector has a weighting that is used to calculate the specificity

Selector	Weight
Type Selector	1
Class Selector	10
ID Selector	100

- When 2 rules have the same specificity the last encountered rule is used.

Specificity Examples

```
div p { color: red; }
p { color: blue; }
```

- `p` has a specificity of 1 (1 HTML selector)
- `div p` has a specificity of 2 (2 HTML selectors)
- Even though p came last p elements inside divs will be coloured red as it is more specific

Other Examples

- `.tree` has a specificity of 10 1 class
- `div p.tree` has a specificity of 12 2 HTML > 1 class
- `#baobab` has a specificity of 100 1 ID
- `body #content .alt p` = 112 1 HTML > 1 ID > 1 Class > 1 HTML

Grouping

- Stylesheets allow you to apply the same rule to multiple selectors

```
selector,selector,...,selector {}
```

```
h1,p.header,#title {color:#f0f}
```

Shorthand Properties

- CSS Syntax is quite open
- Properties can be specified in many ways
- Be aware of the best fit to what you are trying to achieve

```
p {
    border-left-width:4px;
    border-left-style:solid;
    border-left-color:red;
    border-top-width:1px;
    border-top-style:solid;
    border-top-color:red;
    border-right-width:1px;
    border-right-style:solid;
    border-right-color:red;
    border-bottom-width:1px;
    border-bottom-style:solid;
    border-bottom-color:red;
}
```

```
p {
    border : 1px solid red;
    border-left-width : 4px;
}
```

!important

- The !important declaration at the end of a style rule specifies it should take precedence over any other rule regardless of specificity or location

```
<p class=”blue” id=”para”>This text</p>

p{color:red!important;}
p.blue {color:blue;}
#para {color:black}
```

- If more than 1 rule is marked important is applied the last encountered is applied

Javascript Overview

Overview

*"Javascript is a **dynamic**, **weakly typed**, **prototype-based** language with **first-class functions**"*

The design of JavaScript was influenced

- *by Java, which gave JavaScript its syntax,*
- *by Self, which gave JavaScript dynamic objects with prototypal inheritance,*
- *and by Scheme, which gave JavaScript its lexically scoped functions.*
- *JavaScript's regular expressions came from Perl.*

Common Concerns

I would have to learn another language

Javascript is slow

Javascript is temperamental in cross-browser environments

“If you were to take your favourite programming language, strip away the standard libraries and replace them with the DOM, you would hate it.”

Javascript: The Good, The Bad & The Ugly

The Awful Parts

- Global Variables, Scope, Semi-Colon Insertion, Reserved Words, Unicode, Typeof, parseInt, +, Floating Point, NaN, Phony Arrays, Falsy Values, hasOwnProperty, Object

The Bad Parts

- ==, with Statement, eval, continue Statement, switch Fall Through, Blockless Statements, ++/--, Bitwise Operators, Typed Wrappers, new, void

The Good Parts

- 1st Class Functions, Prototypal Inheritance, Object & Array Literal, Closures

The Awful Parts

Global Variables

- Obvious issues with global variables
- All compilation units loaded into global scope
- Implied Global variables are hard to track

```
var a = 'a';

function fun(){
     b = 'b';
     window.c = 'c'
}
alert(a + b + c);
```

Scope

- Block syntax but no block scope

```
function fun(){
     if(x > 0){
          var v = 12;
     }
     alert(v);
}
```

The Awful Parts

Semicolon Insertion

- Semicolons are semi-optional

```
function func(x)
{
     var res = false;
     if(x > 7)
     {
          res = true;
     }

     return
     {
          result:res
     };
}
alert(func(3).result);
```

typeof

- typeof is broken

```
typeof 9            // 'number'
typeof {}           // 'object'
typeof 't'          // 'string'
typeof undefined    // 'undefined'
typeof null         // 'object'
typeof []           // 'object'
typeof new Array()  // 'object'
```

- **This makes type detection unreliable**

The Bad Parts

==/!=

- Perform type coercion and therefore unpredictable unless comparing same types.

```
'' == '0'           // false
0 == ''             // true
0 == '0'            // true
false == 'false'    // false
false == '0'        // true
false == undefined  // false
false == null       // false
null == undefined   // true
' \t\r\n ' == 0     // true
```

- Use ===/!==

+

- **Both adds and concatenates**
- **Determining which ones it will do is tricky**

```
console.log(1 + '1');
console.log(1 + 1);
console.log(1 + 1 + '1' + 3);
console.log(1 + 1 + +'1' + 3);
```

The Bad Parts

eval

- eval is evil

In Closing

- Javascript is very misunderstood and greatly misused.
- Javascript has a lot of awful or bad features
- All of these features are entirely avoidable and a bit of good practise and habit will help avoid them.
- The rest of the course will highlight the good parts and introduce best practise approach to getting good results.

Javascript Fundamentals

Character Set

- Javascript programs are written using the Unicode character set
- Characters represented internally as 2 bytes
- This larger character set should be considered when testing low level functionality

Whitespace & Line Breaks

- Javascript ignores tabs, spaces and new lines between tokens
- However be careful of line breaks when dealing with optional semicolons

Optional Semicolons

- Semicolons are optional provided you have one statement per line

```
x = 'one'
y = 'two'
x = 'one'; y = 'two'
```

- Semicolons automatically inserted at end of lines that are determined to be complete statements.

```
return          return;
true      ⇒     true;
```

Comments

- Both single and multiline comments supported
- C++/Java style comments
- Multiline comments cannot be nested

```
// This is a single-line comment.
/* This is also a comment */
// and here is another comment.

/*
* This is yet another comment.
* It has multiple lines.
*/
```

Identifiers

- First character must be a letter, _ or $
- Remaining characters must be letter, digit, _ or $
- Must not be the same as a word reserved by the language

Reserved Words

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

- The following words are also reserved but never used in the language

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

Data Types and Values

- Javascript defines a simple set of data types
  - 3 primitive types   Numbers, Strings & Booleans
  - 2 trivial types   `null` & `undefined`
  - 1 composite type   Object
  - 1 special type   Function

`null` & `undefined`

- null and undefined are similar but different
- They are equivalent but considered different types

```
null == undefined  // true
null === undefined // false
```

Control Structures

```
if(bork) {
  //...
} else {
  //...
}

while(bork) {
 //...
}

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

for(var element in array_of_elements) {
  //...
}

do {
 //...
} while(bork);
```

```
switch(bork) {
  case 1:
  // if bork == 1...
  case 'whee':
  // if bork == 'whee'...
  case false:
  // if bork == false...
  default:
  // otherwise ...
}

try {
  //...
} catch(err) {
  //...
}
```

Comparison

!=

Non-equality comparison:
Returns true if the operands are not equal to each other.

==

Equality comparison:
Returns true when both operands are equal. The operands are converted to the same type before being compared.

!==

Non-equality comparison without type conversion:
Returns true if the operands are not equal OR they are different types.

===

Equality and type comparison:
Returns true if both operands are equal and of the same type.

Guard & Default

- Guard prevents exceptions being thrown on null object calls

```
var user = null;
var loggedIn = false;
function getUserName(){
     return loggedIn && user.userName
}
```

- Default ensures a default value is returned if a property is null

```
function printArgs(mandatory, optional){
     optional = optional || “default”;
     /* do some stuff */
}
```

`printArgs(1)` is the same as `printArgs(1,’default’)`

Adding JavaScript

- Two ways to include JavaScript on the page

```
<script src="external.js"></script>
<script>
     // code goes here
</script>
```

- JavaScript is executed as it is encountered even if page isn't fully finished.
- Beware of accessing elements before the page is loaded.
- JavaScript should be added as low down the page as possible.
- No need to specify script type.

Numbers

Overview

- No distinction between integers and floating point numbers
- All numbers represented in 64bit floating point format (IEEE 754)
- Floating point arithmetic isn't accurate and can't be relied upon

```
0.1 + 0.2 != 0.3
```

- Integer arithmetic is completely accurate

Integer Literals

- Integers within the extensive +/- 2^{53} can be accurately represented
- Integer literals are in base-10 by default

```
0
5
9989
11
```

Hexadecimal & Octal Literals

- A hexadecimal (base-16) literal begins with 0x
- An octal literal (base-8) begins with 0 but isn't 100% supported
- Never write a literal beginning with 0

```
0xff        // 15*16 + 15 = 255 (base 10)
0xCAFE911

0377        // 3*64 + 7*8 + 7 = 255 (base 10)
```

Floating Point Literals

- Represented using traditional syntax (decimal point), or
- Using exponential syntax

[*digits*][.*digits*][(`E`|`e`)[(`+`|`-`)]*digits*]

- Large numbers can only ever be assumed to be an approximation

```
3.14
0.789
.33333333333333333
6.02e23          // 6.02 x 10^23
1.4738223E-32    // 1.4738223 x 10^-32
```

Infinity

- Attempting to work with numbers outside the range supported will yield a constant

```
Infinity/-Infinity

var result=2;
for (i=1; result!=Infinity; i++){
   result=result*result;
   console.log(i+':'+result);
}
```

- Division by 0 will also yield Infinity

```
console.log(255/0);  // Outputs: Infinity
console.log(-255/0); // Outputs: -Infinity
```

NaN

- Any unsuccessful numeric operation results in a special value

```
NaN   // Not a Number
```

- NaN is always unequal to all other numbers and **itself**.
- If you would like to make sure the result was a number and not the NaN error condition use

```
isNaN(value)
```

- NaN always evaluates to false

Arithmetic Operations

- Typical operations

+	Addition (Concatenation)
-	Subtraction (Unary conversion)
*	Multiplication
/	Division
%	Remainder/Modulus
++	[pre/post]Increment
--	[pre/post]Decrement

parseInt(string, [radix])

- Global function
- Accepts a string and converts it an integer
- Reads from the 1st character to the 1st non numeric character
- Optionally supply a radix to use as the base for conversion. Default depends on string
- Always specify a radix of 10

parseFloat(string)

- Global function
- Extracts floating point numbers from strings
- Same rules as parseInt except will parse past decimal point

The Number Object

- Provides a typed wrapper around literals
- Should never be used directly as it effects comparisons and truth tests

```
var numericLiteral = 0;
var numericObject = new Number(0);
if (numericLiteral) {
    // never executed 0 == false
}
if (numericObject) {
    // always executed object is defined
}
```

- Number methods are copied to numeric literals but properties aren't

Number Object: Properties

- Available ONLY to the Number object itself and not any children or literals

`MAX_VALUE`	Always the largest possible numeric value.
`MIN_VALUE`	Always the smallest possible numeric value.
`NaN`	Value returned on Not A Number errors.
`NEGATIVE_INFINITY`	Value returned on out of range -ve numbers.
`POSITIVE_INFINITY`	Value returned on out of range +ve numbers.

Number Object: Methods

- Available on numeric literals as well as the number object
- Some of these functions are only supported in IE 5.5+/FF1.5+

toExponential([places])

- return a string representation of the number as an exponent
- If you provide a number in the first argument, this method will return only the specified number of decimal places.
- If you are using a numeric literal then you should provide a space between the number and the method. If your number has no decimal you can also add one before calling the method.

toFixed([digits])

- This method attempts to return a string representation of the number as a non-exponent with [digits] numbers after the decimal.
- Handy for working with currency
- IE 5.5 buggy implementation
- *digits* must be in range of 0-20 other wise a RangeError is thrown
- Default is 0

toLocaleString()

- Attempts to format the number to meet the various regional preferences for displaying numbers
- Can result in formatting including commas etc if computer supports it.

toPrecision([precision])

- will return a string with [*precision*] digits after the decimal point.
- *precision* must be a value greater than 0.
- If no p*recision* value is passed, this method behaves like toString().
- Like toFixed(), this method will round up to the next nearest number.
- A *precision* greater than numbers precision will result in inaccuracy

toString([base])

- Outputs the number as a string
- Base is automatically determined in the same manner as a literal, or,
- Supply base as an argument

valueOf()

- simply returns the number as a string.
- Unlike the toString() method, valueOf does not allow you to base conversions.
- The string output is always equal to the number as it's represented in base 10.

Math

- Javascript provides a global object specifically for handling math related work
- Has become increasingly popular with the introduction of canvas
- All methods are accessed in a staic fashion via the constructor, e.g.

```
Math.floor(12.233)
Math.round(21312.22)
Math.sin(y)
```

Strings

Overview

- Can be represented as a literal or an object
- Object methods are available to literals
- Try and avoid the use of the Object over literals
- Passed by value to functions
- Unlike other languages there is no `char` class

String Literals

- String literals can be declared using either single or double quotes

```
var one = "abcdefghijklmnopqrstuvwxyz";
var two = 'abcdefghijklmnopqrstuvwxyz';
```

- To use quotes in a string either mix & match or escape them

```
var one = "'";      var one = "\”";
var two = '”';      var two = '\'';
```

String Properties

- All strings have a single property length

```
var alpha = "abcdefghijklmnopqrstuvwxyz";
var beta = new String("abcdefghijklmnopqrstuvwxyz");

alpha.length  // 26
beta.length       // 26
```

String Methods

- String methods can be classed into two distinct groups

  General Methods: Use strings to perform operations

  RegExp Methods: Use Regular Expression objects to perform operations

These will be explored more in the next section

indexOf(value[, startIndex])

- Returns the index of the first occurrence of the *value*.
- Returns -1 if not found
- Optional starting index to determine origin of search

```
'abc'.indexOf('ab')          //  0
'abc'.indexOf('d')           // -1
'abcabc'.indexOf('cab')      //  2

'abcabc'.indexOf('abc', 1) //  3
'abcabc'.indexOf('cab', 3) // -1
```

replace(pattern, replacement)

- Returns a string with the first instance of *pattern replaced* with *replacement*
- *pattern* can be either a string or regular expression (see next section)

```
“abc”.replace(“a”,  '')                // ab
“abc”.replace(“a”, 'd')                // dbc
“abcabcabc”.replace(“abc”, 'def')   // defabcabc
```

split(delimiter [,limit])

- Very powerful and often used string method
- Splits a string into an array of strings itemised based on the *delimiter* specified
- Optional *limit* argument restricts the size of generated array

```
'a,b,c,d'.split(',')       // ['a','b','c','d']
'a,b,c,d'.split(';')       // ['a,b,c,d']
'a,b,c,d,'.split(',')      // ['a','b','c','d','']

'a,b,c,d'.split(',', 2)    // ['a','b']
```

substring(index [, endIndex])

- Returns part of original string based on *index*
- If no *endIndex* specified returns entire string from *index* to end
- If *endIndex* is less than *index* then arguments are internally reversed

```
'abcdef'.substring(2)      // 'cdef'
'abcdef'.substring(2,5)    // 'cde'
'abcdef'.substring(5,2)    // 'cde'
```

Other Methods

`charAt`	Returns the character at index.
`charCodeAt`	Returns the Unicode Value.
`concat`	Joins Strings (same as +)
`fromCharCode`	Creates a string from the supplied unicode integers.
`lastIndexOf`	Finds last position of a substring.
`slice`	Extracts a substring starting at the index.
`substr`	Like substring except 2nd argument indicates number of characters
`toLowerCase`	Converts the string to lower case
`toUpperCase`	Converts the string to upper case
`valueOf`	See toString()

Regular Expressions

Overview

- Regular Expressions are nothing unique to Javascript
- Javascript regular expressions are a less powerful subset of real Regular Expressions
- Regular Expressions, while more powerful, can be slower for simple tasks compared to straight string manipulation
- A Regular Expression is simply a special object type.

Notation

- Regular Expressions can be declared as either an object or a literal

```
new RegExp(pattern [, flags])    // Object
/pattern/[flags]                 // Literal
```

`new RegExp('abc')` is the same as `/abc/`

`new RegExp('abc','gi')` is the same as `/abc/gi`

Flags

- A string representing control flags can be passed to Regular Expressions
- The three flags that can be set include

`g`	Global – if set returns an array of matches rather than 1st
`i`	Ignore Case – if set ignores case during matching
`m`	Multiline – matching also handles line breaks

- The flags option is just a string of the required combination – `'gim'`, `'gi'`, `'m'`

RegExp Methods

- The RegExp object (and literal) have two methods that can be used.

  RegExp`.exec(string)`, and,
  RegExp`.test(string)`

- exec() applies the regular expression and returns an array of matches (g flag has no effect)
- test() returns true if the RegExp matches at least once, otherwise false

String Methods

- `replace()` and `split()` both support regular expressions as well as strings
- This allows for replace-all functionality

```
String.replace(/pattern/g,value)
```

- `match()` and `search()` both accept RegExps where match returns an array of all matches in the string and search returns the index of the first match

```
"Watch out for the rock!".match(/r?or?/g)   // ['o','or','ro']
"Watch out for the rock!".search(/for/)     // 10
```

- Both match and search are at least 2 times slower than their plain string indexOf counterpart.

Dates

Overview

- A Date in Javascript is just a special class of object
- Can be constructed in a number of ways
- Represented internally as the number of milliseconds from 1st January 1970
- Dates passed and assigned by reference

```
var date1 = new Date()
var date2 = date1

date2.setYear(2006)

console.log(date1,date2)
```

Construction

- There are multiple ways to construct a Date object

```
new Date()
new Date(milliseconds)
new Date(string)
new Date(year, month[, day[, hours[, minutes[, seconds[, ms]]]]])
```

- Calling Date() without new results in a String representation of the current time

Time Zones

- By default created dates are set to the browsers time zone.
- By default dates are output in the users time zone regardless of how they where created (e.g. with a different time zone)
- Javascript Dates have UTC (Coordinated Universal Time) functions (essentailly GMT) for accessing GMT/UTC equivalent dates

Accessing Date Parts

- The Date object provides an API for reading and writing all parts of a date
- Getters and Setters come in 2 flavours
  - Adjusted (to UTC/GMT):

```
getUTC<date_part>
setUTC<date_part>
```

  - Unadjusted (in current time zone)

```
get<date_part>
set<date_part>
```

Date.parse(str)

- Parses a string representation of a date and returns the number of milliseconds since 01/01/1970 00:00:00
- Unparsable dates result in NaN being returned
- Will always accept dates in IETF format e.g.

```
Wed, 18 Oct 2000 13:00:00 EST
```

- Other date formats change between browsers but major browsers all support simple formats based on time zone e.g.

```
dd/mm/yyyy
```

Other Date Functions

`toDateString`	Outputs the date (no time) as a string.
`toGMTString`	Outputs the date adjusted to GMT.
`toString`	Outputs the date as a string
`toTimeString`	Returns the time as a string.
`toUTCString`	Returns the date as a GMT string.
`UTC`	(Static) Date as a UTC timestamp
`valueOf`	See toString()

Objects & Arrays

Objects

Overview

- Composite datatypes made up of other types (properties/functions), or,
- Unordered collection of properties each of which has a name and value
- new Object() notation is officially deprecated
- Created using literal expression – a list of comma separated name/value pairs
- Passed/Assigned by reference

Object Literals

- Comma separated list of name/value pairs enclosed in a block - {..}
- Each property name is a String or Javascript identifier
- Quotes required for reserved words

```
var obj1 = {};
var obj2 = {x:7};
var obj3 = {x:7, y:'66'};
var obj4 = {
    ref       : obj1,
    “class”   : 'object'
}
```

- As functions are 1st class objects they are a valid property of an object

```
var obj = { doStuff:function(x){return x+1;} }
```

Accessing Properties

- Object properties can be accessed in 2 ways
  - dot notation `object.property`
  - via indexed name `object[property_name]`
- The second method is useful when accessing properties dynamically

```
function callFunc(obj,func){
    return obj[func]();
}
```

- Accessing properties that do not exist result in `undefined` being returned

Error Objects

- Javascript defines a number of error objects that are thrown when an error occurs. These include but are not limited to,

```
Error, EvalError, RangeError, ReferenceError,
SyntaxError, TypeError, URIError
```

- Error objects have at least a `message` property. Custom object can be used

```
{message:'An error occurred'}
```

JSON

- JavaScript Object notation
- Subset of Object type
- Composed of only primitive literals, arrays and other JSON objects
- Alternative to XML data transfer
- Much simpler and more inline with JavaScript
- Viable use of `eval()`

```
var result = "{success:true, payload:[2,4,5]}"; // from server
var parsed = eval("(" + result + ")"); // parenthesis fix

console.log(parsed);          // Object success=true payload=[3]
console.log(parsed.success); // true
console.log(parsed.payload[1]);   // 4
```

Arrays

Overview

- An array is a collection of data values, just as an object is
- Where objects have names arrays have numbers or indexes associated with the properties
- Arrays have no real sequence or upper bounds
- Arrays can be created as objects or literals
- The object notation provides no benefits and confuses matters. Use literals.
- Passed/assigned by reference

Array Literals

- Created like an object but with square brackets [] and no need to name the properties

```
var x = [];                      // empty array
var y = [1,2];                   // initial values
var t = [1,'2',true,-1.3];       // mixed types
```

Accessing Values

- Values are accessed using their index and the [] notation from Objects
- If no value exists undefined is returned
- No need to add items sequentially

```
var x = [];
x[2] = 'test';  // x = [undefined, undefined, 'test']
x[3]            // undefined
```

length Property

- Every array has a length property
- Always starts at zero
- Not read-only

```
var x = [1,2,3,4,5,6,7,8,9,10];

x.length     // 10
x.length = 5;
x            // [1,2,3,4,5]
```

Array Methods

`concat`	Joins multiple Arrays
`join`	Joins all the Array elements together into a string.
`pop`	Returns the last item and removes it from the Array.
`push`	Adds the item to the end of the Array.
`reverse`	Reverses the Array
`shift`	Returns the first item and removes it from the Array.
`slice`	Returns a new array from the specified index and length.
`sort`	Sorts the array alphabetically or by the supplied function.
`splice`	Deletes the specified index(es) from the Array.
`toString`	Returns the Array as a string.
`unshift`	Inserts the item(s) to the beginning of the Array.
`valueOf`	see toString

Functions

Overview

- Functions are 1st class objects in Javascript
- Passed around and referenced as variables
- Can be passed as arguments to other functions
- Their scope can be altered

Defining Functions

- There are 2 ways to define functions.

```
/* named function */
function func(named_arguments){
    function_body
}

/* anonymous function */
function(named_arguments){
    function_body
}
```

- The second case can be assigned to a variable name or passed anonymously as an argument to another function
- A function as an argument is called a Lambda function in the functional programming world

Defining Functions

- There is another way to define functions

```
new Function([param1, param2,...paramN], body)
```

- Thinly disguised eval() and should be avoided at all costs.
- Slower, error prone, insecure and look stupid.

Scope

- Javascript is lexically scoped, meaning that at any time in the execution the statement has access to all it's own and ancestors variables
- Two types of scope – functional and global
- No block scope which confuses developers
- Variables defined inside a function are private to that function and its sub-functions or children
- Leaving out var result in a new variable being created in the global scope

Context

- Within each function there is a special variable called `this`
- `this` points the the "owner" of the function
- Usually it is the parent object who becomes `this`
- Populated automatically but can also be manually manipulated

Function Properties & Methods

- Each function, as it is an object, has a number of properties and methods

`arguments`	pseudo-array of arguments passed to the function
`length`	the number of named arguments expected
`constructor`	function pointer to the constructor function.
`prototype`	allows the creation of prototypes.
`apply`	A method that lets you easilly pass function arguments.
`call`	Allows you to call a function within a different context.
`toSource`	Returns the source of the function as a string.
`toString`	Returns the source of the function as a string.
`valueOf`	Returns the source of the function as a string.

arguments

- Functions accept a variable number of arguments
- Similar concept to Java's method overloading
- Unspecified arguments get the assigned undefined type
- They can be named variables or accessed via the arguments object
- The arguments object is not a real array you cannot directly perform array operations on it
- It has 2 properties

`length` Stores the number of arguments

`callee` Pointer to the executing function (allows functions to recurse)

apply()/call()

- Allows you to apply a method of another object in the context of a different object

```
function.apply(thisArg[, argsArray])
function.call(thisArg[, arg1[, arg2[, ...]]]);
```

`thisArg`

Determines the value of this inside the function. If thisArg is null or undefined, this will be the global object.

`argsArray`

An argument array for the object, specifying the arguments with which the function should be called, or null or undefined if no arguments should be provided to the function.

call() takes a variable number of arguments rather than an array

Self Executing Functions

- It is possible to declare an anonymous function and execute it simultaneously
- Self executing functions prevent global namespace pollution and guard isolated code from external side effects
- Simulates block scope

```
(function(){
  var x = 7; // private
  window.x = 'value';
})()
```

Closures

- a **closure** is a function that is evaluated in an environment containing one or more bound variables. When called, the function can access these variables. Or,
- A closure is created when variables of a function continue to exist when the function has returned.
- i.e. a function is defined within another function, and the inner function refers to local variables of the outer function.
- You need to be careful when handling closures as they are a primary cause of memory leaks

Object-Oriented Javascript

Overview

- JavaScript isn’t technically OO but it is object based
- Classless
- Prototypal Inheritance
- Classical Inheritance possible but it essentially forcing the language to behave in a way it wasn’t intended.
- Don’t get too caught up on this topic. Included for clarity

Simple OO

```
var x = {
    firstName: 'James',
    lastName: 'Hughes',
    getName: function(){
            return this.firstName + ' ' + this.lastName;
    }
}

var y = {
    firstName: 'Someone',
    lastName: 'Else',
    getName: function(){
            return this.firstName + ' ' + this.lastName;
    }
}

console.log(x.getName());
console.log(y.getName());
```

Constructor Functions

- Functions always return a value.
- If no return statement specified undefined is returned
- When invoked with `new` functions return an object – `this.` Represents the scope of the newly created object
- Able to modify `this` before it is returned
- Actual return value is ignored
- Be careful omitting the `new` operator

Better Simple OO

```
function Person(first,last){
    this.firstName = first;
    this.lastName = last;
    this.getName = function(){
            return this.firstName + ‘ ‘ + this.lastName;
    }
}

var x = new Person(“James”,”Hughes”);
var y = new Person(“Someone”,”Else”);

console.log(x.getName());
console.log(y.getName());
```

Type Detection

```
function Person(first,last){
    this.firstName = first;
    this.lastName = last;
    this.getName = function(){
            return this.firstName + ‘ ‘ + this.lastName;
    }
}

var x = new Person(“James”,”Hughes”);

console.log(typeof x);                          // object
console.log(x instance Person);                 // true
console.log(x.constructor == Person);           // true
```

`prototype`

- The prototype property is a property of function objects
- Shared base `Object` representing properties and methods of that type
- Is “live” and can be extended to add functionality (metaprogramming)
- Prototype properties appear as properties of the object itself (see hasOwnProperty)
- Basis of the inheritance strategy in JavaScript

prototype

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

var x = new Person();
var y = new Person();

x.setName = function(f,l){
    this.name = f + " " + l;
}

x.setName("James", "Hughes");
y.setName("Other", "Person");

console.log(x.name);
console.log(y.name);

console.log(x.hasOwnProperty("setName"));
```

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

Person.prototype.setName = function(n){
    this.name = n
}

var x = new Person();
var y = new Person();

Person.prototype.setName = function(f,l){
    this.name = f + " " + l;
}

x.setName("James", "Hughes");
y.setName("Other", "Person");

console.log(x.name);
console.log(y.name);

console.log(x.hasOwnProperty("setName"));
```

__proto__

- Objects have a secret link to their constructors prototype __proto__
- Some browsers do not expose this

```
constructor.prototype == __proto__
```

- This creates a constructor chain.
- Property resolution searches up this chain all the way to the base object

Inheritance

```
function Rectangle(){
    this.height = 0;
    this.width = 0;
}

function Rectangle3D(){
    this.depth = 0;
    this.prototype = new Rectangle();
}

Rectangle3D.prototype = new Rectangle();

var r2d = new Rectangle();

r2d.height = 1;
r2d.width = 2;

var r3d = new Rectangle3D();

r3d.height = 3;
r3d.width = 4;
r3d.depth = 5;

console.log(r2d,r3d);
```

```
function extend(parent, child) {
    for (var i in parent) {
        child[i] = parent[i];
    }
    return child;
}

extend(Rectangle,Rectangle3D);

var r2d = new Rectangle();

r2d.height = 1;
r2d.width = 2;

var r3d = new Rectangle3D();

r3d.height = 3;
r3d.width = 4;
r3d.depth = 5;

console.log(r2d,r3d);
```

Other considerations should be made in this function

What if child already has a property?

Should parent properties include prototype properties etc?

Patterns

- Many common design patterns can be used or at least emulated in Javascript
- Access Levels – public, private, protected
- Object Patterns – Singleton, Chaining, Factory, Observer

Public Properties/Methods

```
/* VIA CONSTRUCTOR */
function MyObject(param) {
        this.member = param;
        this.func = function(){
                return member;
        }
}

/* VIA PROTOTYPE */
MyObject.prototype.getMunged = function () {
        return this.member + new Date();
}

var x = new MyObject("arg");
x.member; x.func(); x.getMunged();
```

Private Properties/Methods

```
/* VIA CONSTRUCTOR */
function MyObject(param) {
      this.member = param;

      var privateMember = 7;
      function privFunc(){
            return this.member + privateMember;
      }
}

var x = new MyObject(‘test’);
x.member(); x.privateMember; x.privFunc()
```

Privileged Properties/Methods

```
/* VIA CONSTRUCTOR */
function MyObject(param) {
	this.member = param;

	var privateMember = 7;
	function privFunc(){
		return this.member + privateMember;
	}

	this.priviledged = function(){
		return privFunc()
	}
}

var x = new MyObject('test');

x.member; x.priviledged();
x.privateMember; x.privFunc();
```

Singleton Pattern

```
var MySingleton = (function(){
        var privateProp1 = "one";
        var privateProp2 = "two";

        function privateFunc(){
                return new Date()
        }

        return {
                publicFunc : function(){
                        return privateFunc() + privateProp2;
                },
                publicProp : "three"
        }
})();

MySingleton.publicFunc();
MySingleton.publicProp;
MySingleton.privateFunc();
MySingleton.privateProp;
```

Chaining

```
Array.prototype.method1 = function(){
    this.push("one");
    return this;
}

Array.prototype.method2 = function(){
    this.push("two");
    return this;
}

Array.prototype.method3 = function(){
    this.push("three");
    return this;
}

['old1','old2','old3'].method1().method2().method3();
```

Factory Pattern

```
var XHRFactory = (function(){

    return {
        getInstance : function(){
            try{
                return new XMLHttpRequest()
            }catch(e){
                try{
                    return new ActiveXObject('Msxml2.XMLHTTP')
                }catch(e2){
                    return new ActiveXObject('Microsoft.XMLHTTP')
                }
            }
        }
    }
})()

var xhr = XHRFactory.getInstance();
```

Observer

```
function Observer(){
    var observers = [];

    this.subscribe = function(callback){
        observers.push(callback)
    }

    this.publish = function(msg){
        for(var i=0;i<observers.length;i++){
            observers[i](msg);
        }
    }
}

var eventObserver = new Observer();

eventObserver.subscribe(function(msg){
    console.log("callback triggered: " + msg)
});
eventObserver.subscribe(function(){
    console.log("another callback triggered")
});

eventObserver.publish("a publish message");
```

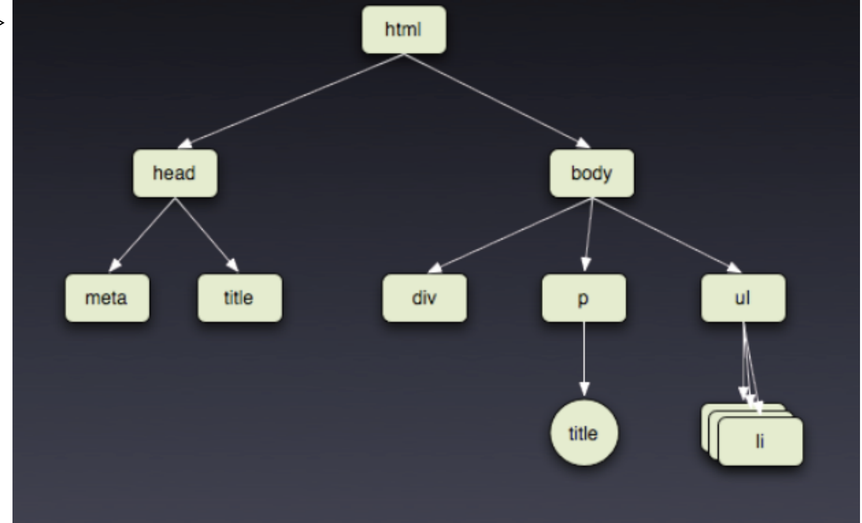
The Document Object Model

Overview

- A language-neutral set of interfaces.
- The Document Object Model is an API for HTML and XML documents. It provides a structural representation of the document, enabling you to modify its content and visual presentation.
- Essentially, it connects web pages to scripts or programming languages.

HTML Document

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                    "http://www.w3.org/TR/html4/strict.dtd">
<html>
     <head>
          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
          <title>ToDo list</title>
     </head>
     <body>
          <div>What I need to do.</div>
          <p title="ToDo list">My list:</p>
          <ul>
               <li>Finish presentation</li>
               <li>Clean up my home.</li>
               <li>Buy a bottle of milk.</li>
          </ul>
     </body>
</html>
```

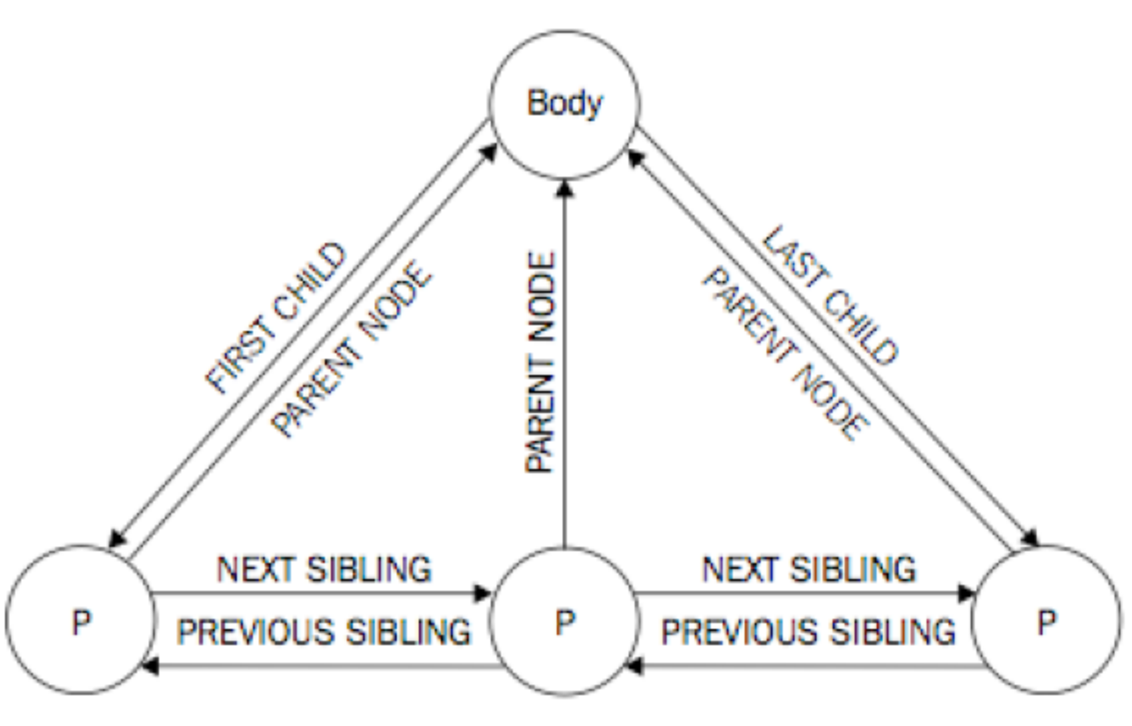


Elements

- HTML is essentially a focused version of XML
- Elements make up the DOM structure
- Nestable
- Can be validated using various DTD's (Strict and Loose)
- DOM Elements provide properties and functions like any other object in JavaScript

Element Attributes

- nodeName
- nodeValue
- nodeType
- parentNode
- childNodes
- firstChild
- lastChild
- previousSibling
- nextSibling
- attributes
- ownerDocument



Node Types

- **<u>NODE_ELEMENT(1) : This node represents an element.</u>**
- NODE_ATTRIBUTE(2) : This node represents an attribute of an element. Note that it is not considered a child of the element node.
- **<u>NODE_TEXT(3) : This node represents the text content of a tag</u>.**
- NODE_CDATA_SECTION(4) : This node represents the CDATA section of the XML source. CDATA sections are used to escape blocks of text that would otherwise be considered as markup.
- NODE_ENTITY_REFERENCE(5) :This node represents a reference to an entity in the XML document.
- NODE_ENTITY(6) : This node represents an expanded entity.
- NODE_PROCESSING_INSTRUCTION(7) : This node represents a processing instruction from the XML document.
- NODE_COMMENT(8) : This node represents a comment in the XML document.
- NODE_DOCUMENT(9) : This node represents an XML document object.
- NODE_DOCUMENT_TYPE(10) : This node represents the document type declaration of the <!DOCTYPE> tag.
- NODE_DOCUMENT_FRAGMENT(11) : This node represents a document fragment. This associates a node or subtree with a document without actually being part of it.
- NODE_NOTATION(12) : This node represents a notation in the document type declaration.

Finding Elements

```
<input type="text" id="message" value="Value"/>

<ul id="list">
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>

var items = document.getElementsByTagName("li");
var msgInput = document.getElementById("message");
```

- You should never have more than one element with the same ID. It allowed but unpredictable (usually the first found though)

DOM Manipulation

```
var item = document.createElement("li");
var text = document.createTextNode(message);

item.appendChild(text);

parent.appendChild(item);
parent.insertBefore(someNode, item);

parent.removeChild(item);
```

InnerHTML

```
parent.innerHTML = parent.innerHTML + ("<li>"+message+"</li>");
```

- Why go through the trouble of creating nodes?
- More efficient
- Easier
- Not part of official standard but fully supported
- But problems lurk

InnerHTML Issues

- Always destroys contents of element even when appending
- Leads to memory leaks (won’t remove event handlers etc)
- Pure String manipulation always has it’s issues
- No reference to the created elements
- Some browsers prevent innerHTML on some elements (IE & <tr> elements for example)

Comparison

```
var item = document.createElement("li");
item.appendChild(document.createTextNode(“item”));
parent.appendChild(item);
```

VS

```
parent.innerHTML += “<li>item</li>”;
```

Document Fragments

- Allows work “off-DOM” on multiple elements
- Document Fragments cannot be appended to DOM
- Child elements automatically appended instead
- Provide a performance enhancement

```
var frag = document.createDocumentFragment();
for( var i = 0, p; i < 10; i++ ) {
        p = document.createElement('p');
  p.appendChild(document.createTextNode(
                 'Paragraph '+(i+1)
        ));
        frag.appendChild(p);
}

document.body.appendChild(frag);
```

JavaScript API

- The window object is represented as a large object with properties and methods for accessing parts of the DOM
- It is huge and for the most part you won't use directly
- http://www.howtocreate.co.uk/tutorials/javascript/javascriptobject

Events

Overview

- Javascript can be said to be event driven
- Events for the heart of all UI interactions
- The event model is the most inconsistent part of Javascript in Cross Browser situations

Netscape Model

- Introduced in Netscape 2
- Inline syntax
- Syntax still used today as it is guaranteed to work
- Cause very tight coupling of model and controller

```
<a href="somewhere.html" onclick="alert('I\'ve been clicked!')">
```

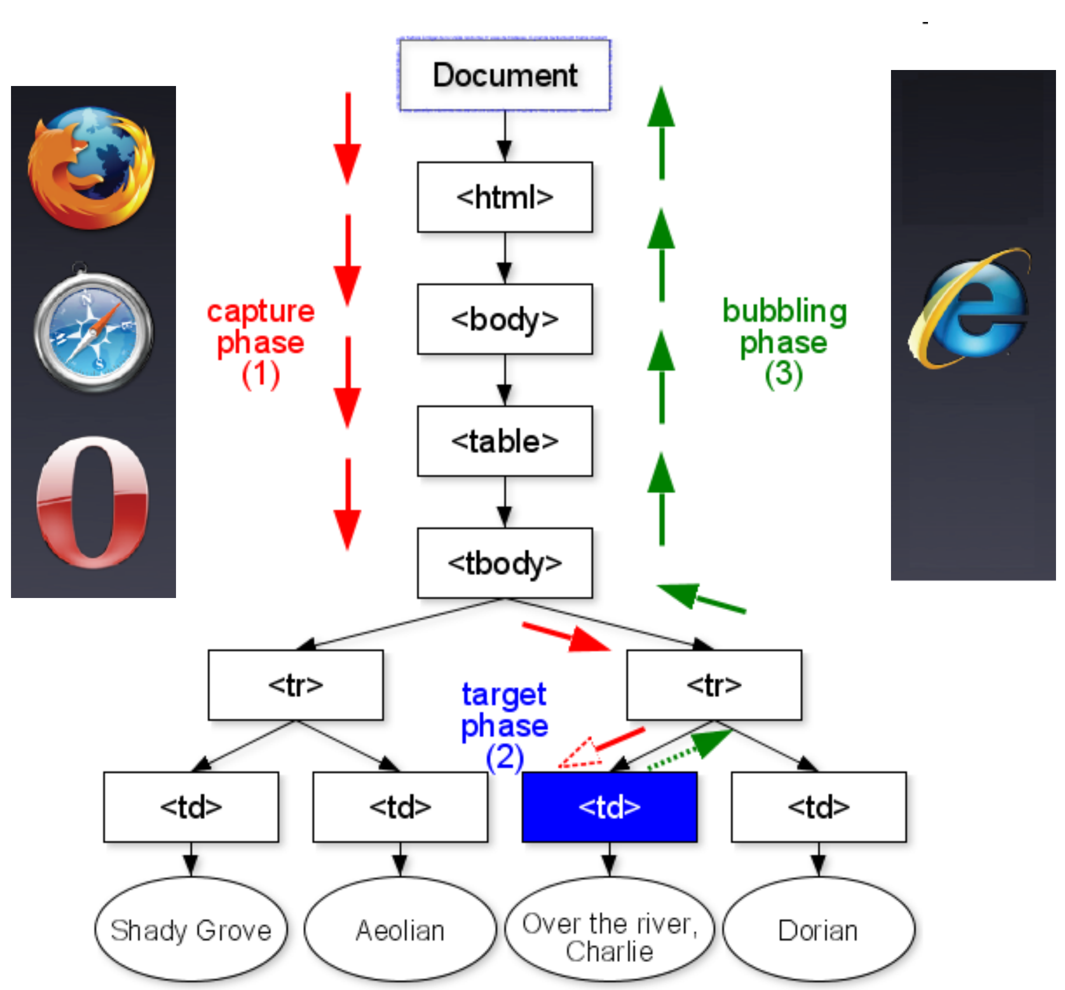
- Ability to reference these events in javascript

```
document.getElementById(‘a-link’).onclick = function(){alert(‘click’)}
document.getElementById(‘a-link’).onclick()  // trigger event
```

Modern Event Models

- Increase in number of available event types
- New registration model (entirely through JavaScript)
- Event Observation Pattern supporting multiple events on the same element
- Inconsistent model across browsers

W3C Event Phases



Event Listener Registration

```
function listen(element, event, callback){
        if (element.addEventListener) {
        /* ALL NON-IE */
      element.addEventListener(event, callback, false);
    } else {
      /* IE */
      element.attachEvent("on" + event, callback);
    }
}
```

- Creates a simple normalized event model across browsers.
- Some edge cases not catered for (browser specific events etc)
- The scope of the callback (`this`) is the element that triggered the event

Available Events

Attribute	The event occurs when...	IE	F	O	W3C
onabort	Loading of an image is interrupted	4	1	9	Yes
onblur	An element loses focus	3	1	9	Yes
onchange	The content of a field changes	3	1	9	Yes
onclick	Mouse clicks an object	3	1	9	Yes
ondblclick	Mouse double-clicks an object	4	1	9	Yes
onerror	An error occurs when loading a document or an image	4	1	9	Yes
onfocus	An element gets focus	3	1	9	Yes
onkeydown	A keyboard key is pressed	3	1	No	Yes
onkeypress	A keyboard key is pressed or held down	3	1	9	Yes
onkeyup	A keyboard key is released	3	1	9	Yes
onload	A page or an image is finished loading	3	1	9	Yes
onmousedown	A mouse button is pressed	4	1	9	Yes
onmousemove	The mouse is moved	3	1	9	Yes
onmouseout	The mouse is moved off an element	4	1	9	Yes
onmouseover	The mouse is moved over an element	3	1	9	Yes
onmouseup	A mouse button is released	4	1	9	Yes
onreset	The reset button is clicked	4	1	9	Yes
onresize	A window or frame is resized	4	1	9	Yes
onselect	Text is selected	3	1	9	Yes
onsubmit	The submit button is clicked	3	1	9	Yes
onunload	The user exits the page	3	1	9	Yes

The Event Object

- With the new event model an event object is exposed
- This gives information on the event
- Inconsistent across browsers
- Access to the event object differs between browsers
  - Global property `window.event` in IE
  - Passed as first argument to callback in other browsers
- Most libraries provide a normalised approach and object

Event Object Properties

Event Delegation

- Possible due to event capture/bubbling
- Single event attached to common parent node (even document)
- Tolerant to dynamic elements and innerHTML updates
- Single function vs many identical functions
- Less memory intensive (think long lists and event rebinding)
- See example (eventdelegation.html)

Ajax Principles

Overview

- AJAX allows your application to talk to the server without entire page refreshes
- AJAX = Asynchronous JavaScript & XML
- Can be synchronous (though often considered extra work)
- Response may not be XML
- Can reduce redundant data transfer effectively cutting network traffic
- Can be achieved without the XMLHttpRequest object

Early AJAX

- This was AJAX before the AJAX buzzword
- Uses a hidden iframe to communicate with the server
- Consistent across the browsers
- Google Mail still uses this approach
- Still used for multipart file uploads

IFRAME Example

```
<html>
    <head>
        <title>Hidden IFRAME Example</title>
    </head>
    <body>

        <iframe name="serverComm" id="serverComm" style="display:none"></iframe>

        <label for="myname">Name:</label><input id="myname" value="" type="text" />
        <button id="answer">Go!</button>

        <script>
            document.getElementById("answer").onclick = function(){

                document.getElementById("serverComm").src="iframecontent.html?name="
                    + document.getElementById("myname").value;

                /* allow screen repaint */
                setTimeout(function(){
                    alert(serverComm.document.getElementById('serverData').innerHTML);
                },0);
            }
        </script>
    </body>
</html>
```

XMLHttpRequest

- Offers greater control over AJAX requests
- Ability to detect errors and other events
- Handled slightly different between IE and other browsers.

XMLHttpRequest Properties

- `readyState` - the status of the request
  - `0` = uninitialized
  - `1` = loading
  - `2` = loaded
  - `3` = interactive (not fully loaded) – useless?
  - `4` = complete
- `responseText` - String value of the returned data
- `responseXML` - DOM-compatible XML value of returned data
- `status` - Numeric status code returned by server.
  Example: 404 for "Not Found"
- `statusText` - The text value of the server status.
  Example: "Not Found"

XMLHttpRequest Methods

- `abort()` - abort the current request
- `getAllResponseHeaders()` - returns as a string all current headers in use.
- `getResponseHeader(headerLabel)` - returns value of the requested header.
- `open(method, URL[, asyncFlag[, userName[, password]]])` - set up a call.
- `send(content)` - Transmit data
- `setRequestHeader(label, value)` - Create or change a header.

XMLHttpRequest Events

- `onreadystatechange` - event handler that deals with state change (`readyState` property)

Getting XMLHttpRequest

```
var XHRFactory = (function(){
    return {
        getInstance : function(){
            try{
                return new XMLHttpRequest()
            }catch(e){
                try{
                    return new ActiveXObject('Msxml2.XMLHTTP')
                }catch(e2){
                            try{
                      return new ActiveXObject('Microsoft.XMLHTTP')
                            }catch(e3){
                              return null;
                            }
                }
            }
        }
    }
})();

var xhr = XHRFactory.getInstance();
```

Sending a Request

```
var xhr = XHRFactory.getInstance();

xhr.open("GET", “iframecontent.html?name=“ + name);
xhr.send(null);
```

- `open`() is sets request asynch by default so `send`() returns instantly
- Monitor the `readystatechange` event to detect responses

Same-Origin-Policy

- Early attempt at reducing XSS attacks
- Ajax Requests can only be made within the same domain otherwise they fail
- <script> tags don't care about same origin policy, so….

```
<script>
        try{
                console.log("before: ",window.$);
        }catch(e){
                console.log(e);
        }

        var x = document.createElement("script");
        x.src="http://prototypejs.org/assets/2008/9/29/prototype-1.6.0.3.js";
        x.onload= function(){
                console.log("after: ",window.$);
        };

        document.body.appendChild(x);

 </script>
```

Detecting Response

```
xhr.open("GET", "iframecontent.html?name=" + name);

xhr.onreadystatechange = function(){/* ASYNCHRONOUS */
        if (xhr.readyState != 4)  {
                return;
        }else{
                alert(xhr.responseText);
        }
}

xhr.send(null);

alert(xhr.responseText); /* SYNCHRONOUS */
```

Detecting Errors

```
xhr.open("GET", "iframecontent.html?name=" + name);

xhr.onreadystatechange = function(){/* ASYNCHRONOUS */
          if (xhr.readyState != 4)  {
                    return;
          }else{
                    if(xhr.status != 200){
                        alert("Error: " + xhr.statusText )
                    }else{
                        alert(xhr.responseText);
                    }
          }
}

xhr.send(null);

if(xhr.status != 200){
          alert("Error: " + xhr.statusText )
}else{
          alert(xhr.responseText); /* SYNCHRONOUS */
}
```

Timing Out

```
xhr.open("GET", “iframecontent.html?name=“ + name);

xhr.onreadystatechange = function(){/* ASYNCHRONOUS */

          var timeout = setTimeout(function(){
                    xhr.abort();
                    alert(“Request Timed Out”);
          },10000);

          if (xhr.readyState != 4)  {
                    return;
          }else{
                    clearTimeout(timeout);

                    if(xhr.status != 200){
                        alert(“Error: ” + xhr.statusText )
                    }else{
                        alert(xhr.responseText);
                    }
          }
}

xhr.send(null);
```

POST Request

```
xhr.open("POST", "iframecontent.html");
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

xhr.onreadystatechange = function(){/* ASYNCHRONOUS */
     if (xhr.readyState != 4)  {
               return;
     }else{
               if(xhr.status != 200){
                   alert("Error: " + xhr.statusText )
               }else{
                   alert(xhr.responseText);
               }
     }
}

xhr.send("name=" + name);
```

Unobtrusive JavaScript

Progressive Enhancement

Rather than hoping for graceful degradation, PE builds documents for the least capable or differently capable devices first, then moves on to enhance those documents with separate logic for presentation, in ways that don't place an undue burden on baseline devices but which allow a richer experience for those users with modern graphical browser software.

Progressive enhancement

Steven Champeon and Nick Finck, 2003

PE in JavaScript

- Build a UI that works without JavaScript
- Use JavaScript to enhance that site to provide a better user experience.
- Should always be able to use the site regardless of device/platform
  - Start with Semantic HTML
  - Layer on some CSS to apply the site's visual design
  - Layer on some JavaScript to apply the site's enhanced behaviour

Why?

- There are legitimate reasons to switch it off
- Some companies strip JavaScript at the firewall
- Some people run the NoScript Firefox extension to protect themselves from common XSS and CSRF vulnerabilities
- Many mobile devices ignore JS entirely
- Screen readers DO execute JavaScript, but accessibility issues mean that you may not want them to

Characteristics of Unobtrusive JavaScript

- No in-line event handlers
- All code is contained in external javascript files
- The site remains usable without JavaScript
- Existing links and forms are repurposed
- JavaScript dependent elements are dynamically added to the page

jQuery

Unobtrusive JavaScript

One Liner

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. ***jQuery is designed to change the way that you write JavaScript.***

Why jQuery over XXX?

- Unlike Prototype and mooTools
  - it doesn’t clutter your global namespace
- Unlike YUI it’s succinct
  - YAHOO.util.Dom.getElementsByClassName()
- Unlike Dojo
  - the learning curve is hours, not days
- Unlike ExtJS
  - the license isn’t an issue
  - the foot print is negligible
- Adopted by Microsoft & Nokia as their core client side library
- Highly active and passionate community
- Well documented (http://api.jquery.com)
- Structured to be extensible

jQuery Characteristics

- Minimal namespace impact (one symbol)
- Focus on the interaction between JavaScript and HTML
- (Almost) every operation boils down to:
  - Find some elements
  - Do things with them
- Method chaining for shorter code
- Extensible with plugins

Non-Polluting

- Almost everything starts with a call to the `jQuery()` function
- Since it's called so often, the `$` variable is set up as an alias to `jQuery`
- However, if you're also using another library (such as Prototype) you can revert to the previous $ function with

```
jQuery.noConflict();
```

No More `window.onload`

- Almost all pages require some sort of -”when page is loaded do x, y and z”

```
window.onload = function(){ /* code */ }
```

- Usual inline event problems
- Code isn’t executed until entire document (including images) is loaded not just DOM
- jQuery permits code to be executed as soon as the DOM is ready
- Improves perceived responsiveness

```
$(document).ready(function(){
   /* CODE HERE */
});
```

```
$(function(){
   /* CODE HERE */
});
```

`jQuery()/$()`

- Overloaded behaviour depends on type of argument
  - Listen for DOM Ready events
  - Select elements from the DOM
  - Enhance a DOM element
  - Create new DOM Nodes
- Usually returns another jQuery object (supports chainability) which is essentially an array type object enhanced with jQuery functionality

Selectors

- jQuery/$ accepts a CSS selector and returns a collection of matching elements (as a jQuery object)
- CSS1,2 & 3 selectors supported (more so than current browser implementations!)
- Custom selectors available
- Available selectors can be extended
- Support for very complex selectors

CSS Selectors

```
$('*')
$('#nav')
$('div#intro h2')
$('#nav li.current a')
```

CSS 2 & 3 Selectors

```
a[rel]
a[rel="friend"]
a[href^="http://"]
ul#nav > li
#current ~ li (li siblings that follow #current)
li:first-child
li:last-child
li:nth-child(3)
```

Custom Selectors

- :first, :last, :even, :odd
- :header
- :hidden, :visible
- :input, :text, :password, :radio, :submit
- :checked, :selected, :enabled, :disabled
- div:has(a), div:contains(Hello), div:not(.entry)
- :animated

Adding Custom Selectors

- If jQuery doesn't have it, add it

```
/* v1.3+ approach*/
$.extend(jQuery.expr.filters, {
    hasTooltip: function(e,i,m,a){
            return !!$(e).attr("title");
    }
});

/* pre 1.3 approach */
$.expr[':'].hasTooltip = function(e,i,m,a){
    return !!$(e).attr("title");
}

$('div:hasTooltip')
```

Support for passing args

jQuery Collections

- The jQuery/$ function returns a jQuery Collection object
- You can call treat it like an array

```
$('div.section').length
$('div.section')[0]
$('div.section')[2]
```

- You can call methods on it:

```
$('div.section').size()
$('div.section').each(function() {
        console.log(this);
});
```

Chainability

- Almost all jQuery functions (unless unnecessary) return a jQuery collection object
- Ability to chain code making it simpler and shorter

```
$('div.section').addClass('foo').hide();
```

- Complex collection filtering supported rather than having to perform multiple selection

```
$(‘tbody’).find(‘tr:odd’)
                    .css(‘background-color:#eee’)
                    .end()
                    .find(‘tr:even’)
                    .css(‘background-color:#aaa’)
```

jQuery Methods

- jQuery provides methods to operate on the returned collections
- These can be grouped into 4 main types
    1. **Introspectors** - return data about the selected nodes
    2. **Modifiers** - alter the selected nodes in some way
    3. **Navigators** - traverse the DOM, change the selection
    4. **DOM modifiers** - move nodes within the DOM

Introspectors

```
$('div:first').attr('title')
$('div:first').html()
$('div:first').text()
$('div:first').css('color')
$('div:first').is('.entry')
```

Modifiers

```
$('div:first').attr('title', 'The first div')
$('div:first').html('New <em>content</em>')
$('div:first').text('New text content')
$('div:first').css('color', 'red')
```

Bulk Modifiers

```
$('a:first').attr({
    title: 'First link on the page',
    href : 'http://www.kainos.com/'
});

$('a:first').css({
    color: 'red',
    backgroundColor: 'blue'
});
```

Accessor Pattern

`$(selector).attr(name)`	gets
`$(selector).css(name)`	gets
`$(selector).attr(name, value)`	sets
`$(selector).css(name, value)`	sets
`$(selector).attr({ object })`	sets in bulk
`$(selector).css({ object })`	sets in bulk

Style Modifiers

```
$(selector).css(...)
$(selector).addClass(class)
$(selector).removeClass(class)
$(selector).hasClass(class)
$(selector).toggleClass(class)
```

Dimensions

```
$(selector).height()
$(selector).height(200)
$(selector).width()
$(selector).width(200)
var offset = $(selector).offset()
  offset.top
  offset.left
```

Navigators - Finding

```
$('h1').add('h2')

$('div:first').find('a')

$('a:first').children()

$('a:first').children('em')

$('a').parent()

$('a:first').parents()
```

```
$('a:first').siblings()

$('h3').next()

$('h3:first').nextAll()

$('h3').prev()

$('h3').prevAll()

$('a:first').contents()
```

Navigators - Filtering

```
$('div').eq(1) // gets second

$('div').filter('.entry')

$('div').filter(function(i) {
   return this.title == 'foo'
})
```

```
$('div').not('.entry')

$('div').slice(1, 3) // 2nd,3rd

$('div').slice(-1)   // last
```

DOM Modifiers

```
els.append(content)

content.appendTo(els)

els.prepend(content)

content.prependTo(els)

els.after(content)

els.before(content)

content.insertAfter(els)

content.insertBefore(els)

els.wrapAll('<div /'>)

els.wrapInner('<div /'>)

els.empty()

els.remove()
```

DOM Construction

- Internally handles orphaned nodes, events and data

```
var p = $('<p id="foo" />');      // create node
p.text('Text');                   // update node
p.appendTo(document.body);        // append to DOM

/* Or as a oneliner */
$('<p id="foo" />').text('Text').appendTo(document.body);
```

Events

```
$('a:first').bind('click', function() {
   $(this).css('backgroundColor' ,'red');
   return false;
});

$('a:first').click(function() {
   $(this).css('backgroundColor' ,'red');
   return false;
});
```

Event Object

- jQuery passes a normalized W3C Event object to all event callbacks
- Always passed to callback – no need to check for `window.event`

Event Object Attributes

- `type` - Describes the nature of the event.
- `target` - Contains the DOM element that issued the event
- `currentTarget` - The current DOM element within the event bubbling phase. This attribute will always be equal to the *this* of the function
- `pageX/Y` - The pageX/Y property pair returns the mouse coordinates relative to the document.
- `result` - Will contain the last value returned by an event handler (that wasn't undefined).
- `timeStamp` - The timestamp (in milliseconds) when the event was created.

Event Object Methods

- `preventDefault()` - Prevents the browser from executing the default action.
- `isDefaultPrevented()` - Returns whether preventDefault() was ever called on this event object.
- `stopPropagation()` - Stops the bubbling of an event to parent elements, preventing any parent handlers from being notified of the event.
- `isPropagationStopped()` - Returns whether stopPropagation() was ever called on this event object.
- `stopImmediatePropagation()` - Keeps the rest of the handlers from being executed.
- `isImmediatePropagationStopped()` - Returns whether stopImmediatePropagation() was ever called on this event object.

Triggering Events

```
$('a:first').trigger('click');

$('a:first').click();
```

Supported Events

- `blur()`
- `change()`
- `click()`
- `dblclick()`
- `error()`
- `focus()`
- `keydown()`
- `keypress()`
- `keyup()`
- `load()`
- `mousedown()`
- `mouseover()`
- `mouseup()`
- `resize()`
- `scroll()`
- `select()`
- `submit()`
- `unload()`

Advanced Events

```
$('a:first').unbind('click');

$('a:first').unbind();

$('a').live('click‘, function(){}); // delegation

$('a:first').one('click', function() { })

$('a:first').toggle(func1, func2);

$('a:first').hover(func1, func2);
```

Custom Events

```
/* SUBSCRIBE */
$(window).bind('mail-recieved', function(event, mail) {
    alert('New e-mail: ' + mail);
})

/* PUBLISH */
$(window).trigger('mail-recieved', “New Mail Content”)
```

Ajax

- Simple:

```
$('div#news').load('/news.html');
```

- Complex:

```
$.ajax(options)
$.get(url, [data], [callback])
$.post(url, [data], [callback], [type])
$.getJSON(url, [data], [callback])
$.getScript(url, [data], [callback])
```

Ajax Events

- Two types of Ajax events
  - Local Events: These are callbacks that you can subscribe to within the Ajax request object
  - Global Events: These events are broadcast to all elements in the DOM, triggering any handlers which may be listening.

Local Events

```
$.ajax({
    beforeSend: function(){
        // Handle the beforeSend event
    },
    complete: function(){
        // Handle the complete event
    }
});
```

Global Events

```
/* PER SELECTOR */
$("#loading").bind("ajaxSend", function(){
    $(this).show();
}).bind("ajaxComplete", function(){
    $(this).hide();
});

/* GLOBALLY */
$.ajaxSetup({
            ajaxStart : function(xhr){
                    $("#loading").show()
            },
            ajaxStop : function(xhr){
                    $("#loading").hide()
            }
});
```

Event Order

- **ajaxStart** (Global Event)
  This event is broadcast if an Ajax request is started and no other Ajax requests are currently running.
  - **beforeSend** (Local Event)
    This event, which is triggered before an Ajax request is started, allows you to modify the XMLHttpRequest object (setting additional headers, if need be.)
  - **ajaxSend** (Global Event)
    This global event is also triggered before the request is run.
  - **success** (Local Event)
    This event is only called if the request was successful (no errors from the server, no errors with the data).
  - **ajaxSuccess** (Global Event)
    This event is also only called if the request was successful.
  - **error** (Local Event)
    This event is only called if an error occurred with the request (you can never have both an error and a success callback with a request).
  - **ajaxError** (Global Event)
    This global event behaves the same as the local error event.
  - **complete** (Local Event)
    This event is called regardless of if the request was successful, or not. You will always receive a complete callback, even for synchronous requests.
  - **ajaxComplete** (Global Event)
    This event behaves the same as the complete event and will be triggered every time an Ajax request finishes.
- **ajaxStop** (Global Event)
  This global event is triggered if there are no more Ajax requests being processed.

Animation

- jQuery has built in effects:

```
$('h1').hide('slow');
$('h1').slideDown('fast');
$('h1').fadeOut(2000);
```

- Chaining automatically queues the effects:

```
$('h1').fadeOut(1000).slideDown()
```

Animation

- You can specify custom animations

```
$("#block").animate({
      width:              "+=60px",
      opacity:            0.4,
      fontSize:           "3em",
      borderWidth:        "10px"
}, 1500);
```

Plugins

- jQuery is extensible through plugins, which can add new methods to the jQuery object
  - Form: better form manipulation
  - UI: drag and drop and widgets
  - ... many more

Plugin Development

```
/* Selector Plugin */
jQuery.fn.log = function(message) {
    if (message) {
            console.log(message, this);
    } else {
    console.log(this);
    }
    return this;
};

$(document).find('a').log("All <a>'s").eq(0).log()

/* Static Plugin */
$.hideLinks = function() {
    return $('a[href]').hide();
}

$.hideLinks()
```

Data Cache

- Attaching data directly to DOM nodes can create circular references and cause memory leaks
- jQuery provides a data() method for safely attaching information

```
$('div:first').data('key', 'value');
console.log($('div:first').data('key'));
$('div:first').removeData('key');
```

Utility Methods

- A simple set of utility methods are provided to help with common tasks
- Some work directly on a jQuery collection
- Some are called statically

Utility Methods

```
$('a').map(function(i,e){
    return this.href;
})

$('a').get()

$('a').each(function(){
    if(this.href == '#'){
        return false; // stop
    }

    $(this).attr(
        "title", "external"
    );
})

$.map([1,2,3], function(i,e){
    return this + i;
}) // [1,3,5]
```

```
$.each([1,2,3], function(){
    console.log(this)
})

$.merge([1,2], [4,7]) //[1,2,4,7]

$.unique([1,2,1])     // [1,2]

$.grep(array, callback, invert)

$.makeArray(arguments)

$.inArray(1, [1,2,3])

$.extend(deep, target, object1,
    objectN)
```

Performance

A Pragmatists Approach

Overview

- Performance is a key factor in Javascript
- Inconsistent client specs
- Everything is being pushed to the front end leaving your server as a set of glorified web services
- Browsers being pushed beyond their limits
- JavaScript is parsed and interpreted each time.
- Some simple rules

Be Lazy

Write less code

- Initial <u>parsing</u> of JavaScript is often a major bottleneck
  - No JIT, no cached object code, interpreted every time
- Can't rely on browser caching to excuse large code size
  - Yahoo study: surprising number of hits with empty cache
  - Frequent code releases → frequently need to re-download
- More code = more to download, execute, maintain, etc.
  - Ideal for large AJAX apps is <500K JS uncompressed

Write Less Code

- Minimize the JavaScript code you send down
  - Minify = good, obfuscate = not much better
  - Strip debug / logging lines (don't just set log-level = 0)
  - Remove unnecessary OOP boilerplate
    - Get/Set functions don't actually protect member vars! etc.
- Minimize dependency on third-party library code
  - Lots of extra code comes along that you don't need
  - Libraries solve more general problems → use like scaffolding

Be Responsive

Minimize Perceived Load Time

- Put CSS at the top of your page and JS at the bottom
- Draw major placeholder UI with “loading…” first. Offer constant feedback.
- Load / draw your application progressively (lazy, on-demand)

Example

```
<html>
   <head></head>
   <body>
        <div id="msg">Loading 1st Library</div>
        <script>
               var msg = document.getElementById('msg');
        </script>
        <script src="firstLibrary.js"></script>
        <script>
               msg.innerHTML = "Loading Second Library";
        </script>
        <script src="secondLibrary.js"></script>
        <script>
               msg.innerHTML = "Complete...";
        </script>
   </body>
</html>
```

Yield

- Always want to show a quick response acknowledgement
  - But browser often doesn't update UI until your code returns!
- Solution: do minimum work, use setTimeout(0) to yield
  - Use closures to chain state together with periodic pauses
  - Use `onmousedown` instead of `onclick` (~100msec faster!) but be aware of repercussions
  - http://josephsmarr.com/oscon-js/yield.html?9999999

Cache Back End Responses

- All data requests should go through data-manager code
  - Request as needed and cache results for subsequent asks
  - Requesting code always assumes async response
- Use range caches → only fill in missing pieces
  - Ideal for partial views into long lists of data
- Balance local updates vs. re-fetching from APIs
  - Do the easy cases, but beware of too much update code
  - Worst case = trash cache and re-fetch = first-time case

Be Pragmatic

Be Aware of Browsers Strengths

- Avoid DOM manipulation; use **`innerHTML`** and **`array.join(“”)`**
- Avoid dynamic CSS-class definitions & CSS math
- Avoid reflow when possible (esp. manually on browser resize)
- Avoid memory allocation (e.g. string-splitting)
- Do DOM manipulation off-DOM, then re-insert at the end

Cheat When You Can

- Use IDs when reasonable
  - Finding by class / attaching event handlers is slow
  - Protect modularity only when needed (e.g. widgets)
- Directly attach **`onclick`**, etc. handlers instead of using event listeners where appropriate
- Use fastest find-elems available when you need to scan the DOM (don't rely on general-purpose code) - sizzle

Inline Initial API Calls & HTML

- Tempting to load blank page and do everything in JavaScript
  – Have to redraw UI dynamically; don't want two copies of UI code
- Problem: initial load is usually too slow
  – Too many round-trips to the server; too long before initial UI shows up
- Solution: if you have to do it every time, do it statically
  – Save out initial API responses in web page
  – Use data-manager to hide pre-fetching (can change your mind later)
  – Download initial HTML in web page

Be Vigilant

Be Aware of Aliases

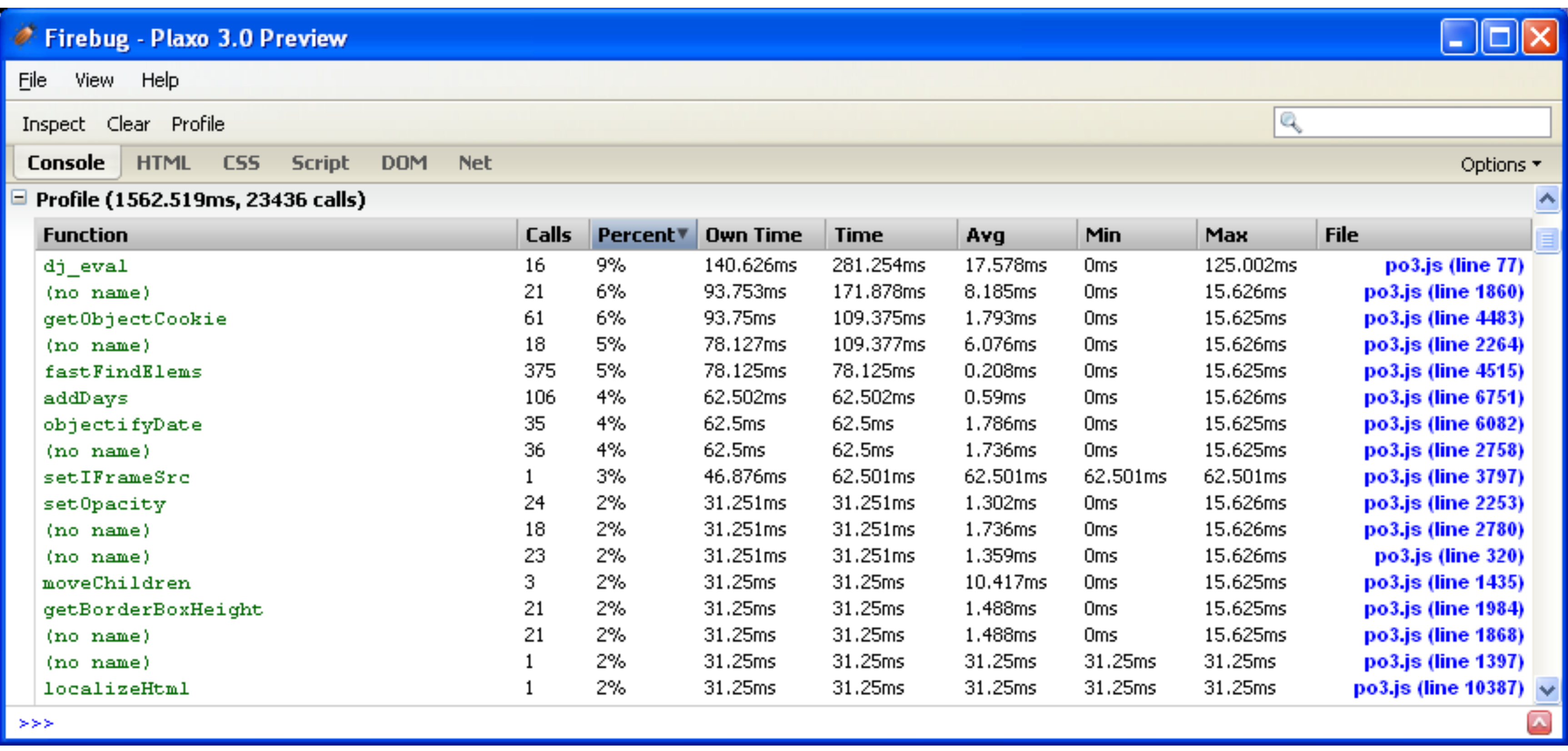
```
var getEl = document.getElementById;

function checkEl(id){
     if(getEl(id) && !getEl(id).checked){
             getEl(id).checked = true;
     }
}

checkEl('somelist');
```

- For that one function call we have **3** calls to getElementById()!
- Common mistake when using 3rd party libraries (Prototype, jQuery make it easy to do this). Not always this obvious.

Profile

- Bottlenecks abound and are usually not obvious
  - Use firebug's profiler
  - Use timestamp diffs and alerts
  - Comment-out blocks of code
- Measure with a consistent environment
  - Browsers bog down → always restart first
  - Try multiple runs and average (and don't forget the cache)
- If possible define responsiveness rules from the start.

Firebug Profiler



Function	Calls	Percent	Own Time	Time	Avg	Min	Max	File
dj_eval	16	9%	140.626ms	281.254ms	17.578ms	0ms	125.002ms	po3.js (line 77)
(no name)	21	6%	93.753ms	171.878ms	8.185ms	0ms	15.626ms	po3.js (line 1860)
getObjectCookie	61	6%	93.75ms	109.375ms	1.793ms	0ms	15.625ms	po3.js (line 4483)
(no name)	18	5%	78.127ms	109.377ms	6.076ms	0ms	15.626ms	po3.js (line 2264)
fastFindElems	375	5%	78.125ms	78.125ms	0.208ms	0ms	15.625ms	po3.js (line 4515)
addDays	106	4%	62.502ms	62.502ms	0.59ms	0ms	15.626ms	po3.js (line 6751)
objectifyDate	35	4%	62.5ms	62.5ms	1.786ms	0ms	15.625ms	po3.js (line 6082)
(no name)	36	4%	62.5ms	62.5ms	1.736ms	0ms	15.625ms	po3.js (line 2758)
setIFrameSrc	1	3%	46.876ms	62.501ms	62.501ms	62.501ms	62.501ms	po3.js (line 3797)
setOpacity	24	2%	31.251ms	31.251ms	1.302ms	0ms	15.626ms	po3.js (line 2253)
(no name)	18	2%	31.251ms	31.251ms	1.736ms	0ms	15.626ms	po3.js (line 2780)
(no name)	23	2%	31.251ms	31.251ms	1.359ms	0ms	15.626ms	po3.js (line 320)
moveChildren	3	2%	31.25ms	31.25ms	10.417ms	0ms	15.625ms	po3.js (line 1435)
getBorderBoxHeight	21	2%	31.25ms	31.25ms	1.488ms	0ms	15.625ms	po3.js (line 1984)
(no name)	21	2%	31.25ms	31.25ms	1.488ms	0ms	15.625ms	po3.js (line 1868)
(no name)	1	2%	31.25ms	31.25ms	31.25ms	31.25ms	31.25ms	po3.js (line 1397)
localizeHtml	1	2%	31.25ms	31.25ms	31.25ms	31.25ms	31.25ms	po3.js (line 10387)

Conclusion

- Stop pushing needless tasks to the browser
- Performance has a higher priority (over other design factors) on the client than on the server
- Remember
  - Be lazy
  - Be responsive
  - Be pragmatic
  - Be vigilant

Tools

JSLint

- JavaScript program that looks for problems in JavaScript programs.
- Identifies common mistakes and overlooked issues
  - Accidental global declaration
  - Accidental closures
  - Missing Semi-colons
- Produces a report of all global declarations, closures, unused variables.
- Very strict.
- Will hurt your feelings.

YUI Compressor

- Compresses JavaScript and CSS
- Does it in a safe manner (compared to Packer which has broken certain scripts)
- Produces one if the best compression ratios
- Bundles JSLint with it
- Java based – build tool plugins

JSDoc

- JavaScript Documentation Tool
- Modelled on JavaDoc
- Should be used in conjunction with compression tools to avoid code/comment bloat

```
/**
 * Function description
 *
 * @param {String} paramName This is a string
 * @returns The radius of this circle
 */
```

QUnit

- Javascript based Unit Testing aimed specifically at testing jQuery based code but can effectively be used with any JavaScript code
- Pages represent testsuites
- Produces interactive reports of tests

Best Practise

Some useful tips

Prefer Literals

Use This…

```
var o = {};

var a = [];

var re = /[a-z]/gmi;

var fn = function(a, b){
  return a + b;
}
```

Over This…

```
var o = new Object();

var a = new Array();

var re = new RegExp('[a-z]', 'gmi');

var fn = new Function(
    'a, b','return a+b‘
);
```

Namespace

- Avoid polluting global namespace
- Avoid conflicts

Encapsulate Code Blocks

- Avoids pollution of global namespace
- No block scope spells trouble

Avoid Extending Native Objects

- Firefox.next will prevent overriding native functions
- Unpredictable
- Requires rework when conflicts are hit (e.g. Prototype vs Firefox)
- 3rd Party Libraries might do the same

Keep Function Style Consistent

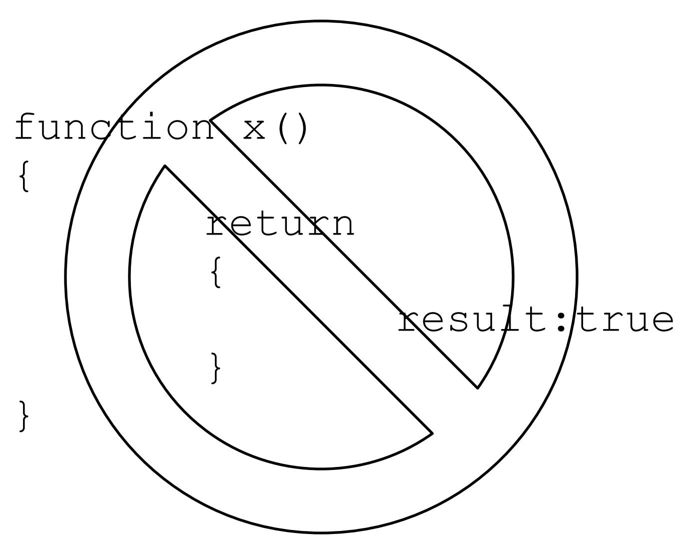
- Function signatures should describe what it expects (where appropriate)
- Mandatory arguments should be named.
- Optional arguments should be accepted as a hash that can be applied over defaults

```
function func(mand1, mand2, opts){
        var options = $.extend({}, {
                opt1 : 'test',
                opt2 : true
        }, opts);
        /* do functiony stuff */
}
```

Keep Braces Consistent

- Avoids uncontrolled semi-colon insertion

```
function x(){
        return {

        result:true
        }
}
```



Use jQuery

- Or at least a library
- Avoid bloated libraries unless necessary
- Keep plugins to a minimum

Put JavaScript at the Bottom

- Page rendering is blocked until JavaScript is executed.
- Blocks parallel downloads
- Screen may appear jumpy or laggy
- Use DEFER only in emergency and conditions permit

Put CSS at the Top

- CSS is applied progressively
- Avoids the flash of unstyled content
- Required by HTML specification!

Lint Your Code

- www.jslint.com – will hurt your feelings!

Compress Your Code

- JSMin
- Packer
- YUI Compressor
- GZIP
- Be careful when using packer inline with GZIP

Use CSS Sprites

- Reduces HTTP request count

Remove Duplicated Scripts

- A review of the ten top U.S. web sites shows that two of them contain a duplicated script

Use GET over POST

- In most browsers POST is a 2 step process (send headers first then content)
- Near doubling of traffic
- Use GET where security isn’t a concern

Delegate

- When dealing with many elements delegate events rather than binding the same function to each element
- Proofs page from dynamic DOM manipulation or innerHTML updates.

Use External Files

- Externalizing JS and CSS can provide faster pages
- More likely to be cached if external (change less)
- Future browsers will compile JavaScript (Google Chrome). Easier to do if external

Resources

Compression Tools

JSMin
http://crockford.com/javascript/jsmin

YUICompressor http://developer.yahoo.com/yui/compressor/

Packer
http://dean.edwards.name/weblog/2007/04/packer3/

CompressorRater
http://compressorrater.thruhere.net/

jQuery

API

http://api.jquery.com

Plugins

http://plugins.jquery.com/

UI

http://ui.jquery.com/

Debugging/Logging Tools

Firebug
http://getfirebug.com/

Firefox Developer Toolbar
http://chrispederick.com/work/web-developer/

Firebug Lite
http://getfirebug.com/

BlackbirdJS
http://code.google.com/p/blackbirdjs/

Microsoft Script Debugger
http://www.microsoft.com/downloads/details.aspx?familyid=2f465be0-94fd-4569-b3c4-dffdf19ccd99&displaylang=en

Code Checking Tools

JSLint

http://www.jslint.com/

Books