# Introduction to JavaScript

Joshua McNeese
Develop Intelligence

## Introductions

- Who am I?
  - Currently VP of Technology for HouseParty Inc
  - Worked in the internet industry for 20 years
  - Web application developer for 15 years
  - Frontend JS development for 10 years
  - Backend JS development for almost 5 years
  - Technical instructor/trainer for about 3 years
  - Live in Colorado, USA

## Introductions

- Who are you?
- What do you want to get out of this class?
- Tell me about your current development environment/process

# How this class works…

- Class starts whether everyone is here or not
- Break in the morning, lunch around noon (for an hour or so), break in the afternoon
- Class is done when we cover everything or our brains hurt too much to continue
- I'm flexible!
  - Ask questions if something isn't clear
  - Tell me if you already know something and we'll skip it
  - We can go hands-on at any point if something is unclear, just let me know
- There's too much to cover in too little time, but we'll try

# IDEs & Editors

- Use whatever you feel most comfortable with, but I recommend:
  - WebStorm http://www.jetbrains.com/webstorm
  - SublimeText http://www.sublimetext.com
  - VIM http://www.vim.org
  - Notepad++ http://notepad-plus-plus.org

# devdocs.io

- Online, searchable application for various programming language documentation
- http://devdocs.io

## Goals for this class

- Become familiar with JavaScript's:
  - Lexical structure
  - Data types
  - Operators
  - Control structures
  - Functions
  - Objects
- Learn about:
  - JavaScript in the browser
  - DOM API
  - AJAX / XHR
- Be able to:
  - Build a webpage that uses JavaScript to respond to user interaction, and dynamically handle data from a server

# PDF for today

- In case you didn't quite catch something, or can't see the screen well:
  - http://bit.ly/1Jx94D5

## Day 1

- About JavaScript
- History & Today
- Lexical structure
- Data types
- Operators
- Control structures
- Functions

## What is JavaScript?

- ECMAScript
- Interpreted
- Case-sensitive C-style syntax
- "Loosely" typed
- Fully dynamic
- Single-threaded event loop
- Unicode (UTF-16, to be exact)
- Prototype-based
- Multi-paradigm
- Kind of weird

## History

- 1995 - Netscape wanted interactivity like HyperCard w/ Java in the name
- Designed & built in 10 days by Brendan Eich as "Mocha", released as "LiveScript"
- Combines influences from:
  - Java, "Because people like it"
  - SmallTalk, prototypal

## Versions

- LiveScript
  - Released in 1995 in Netscape Navigator 2.0
  - Also released as a server-side runtime in Netscape Enterprise Server
- 1.0 – Supported in IE3.0
- ES2/1.3 – Submitted to ECMA for standardization
- ES3/1.5
  - Released in 1999 – in all browsers by 2011
  - IE6-8
- ES5/1.8
  - Released in 2009
  - IE9+
  - http://kangax.github.io/compat-table/es5/
- ES6 is finalized
- ES7 in progress
- Can convert ES6/7 down to ES5, via "transpilers"

## Why JavaScript?

- JavaScript Everywhere™!
- Despite the shortcomings, it's pretty awesome
  - Very expressive
  - Very flexible (that multi-paradigm thing)
  - Lightweight
- The language of the web
  - The browser
  - Client-side frameworks
  - A server and command line via Node.js
  - Beginning to dominate the entire software stack
- Easy to learn, harder to master

## Obligatory Hello World

- In a browser, open a developer console and type:

```
console.log('Hello World!');
```

## Browser Console

- Use your browser developer tools to access its JavaScript console
  - The browser's "`console`" is a REPL
- All major browsers are converging to the same API for console
- Can use it to set breakpoints
  - Let you see scoped variables and context
  - Can set a conditional break-point
- This is where we'll be working; follow along!
  - console.log() => sprintf()

## Language fundamentals

- Lexical structure
- Comments
- Data types
- Variables
- Constants
- Operators
- Control structures

## Lexical structure

- Being a C/C++ like language:
  - Instructions are called **statements** and are separated by a semicolon
  - Spaces, tabs and newlines are called **whitespace**
    - Whitespace generally doesn't matter*
  - Blocks are wrapped with curly braces { }

## Comments

- Follow C/C++ conventions:

```
/*
 span
 multiple
 lines
 */

// comment until the end of the line, or end
// of current statement, whichever is first

/**
 * docblock style can be consumed by third-party
 * packages like jsdoc
 */
```

## Variables

- Must start with a letter, underscore (_), or dollar sign ($)
- Subsequent characters can be alphanumeric, _ or $
- Case-sensitive!
- Unicode characters are supported

```
var 🍔 = 'burger';
```

## Declaring variables

- With the keyword **var**
- One by one:

```
var foo = 'bar';
var thing1 = 2;
```

- Or in sequence:

```
var a = 1, b = 2;
```

- Omitting the **var** keyword creates a global variable

```
stuff = [1,2,3];
```

## Variable scope

- Any variable declared outside of a function, or by omitting the **var** keyword within a function, creates a global variable
- Variables created within a function are called "local variables" or "function variables"

# Variable hoisting

- All variables declaration are "hoisted" to the top of their scope, regardless of where they were defined
  - Initialization is *not* hoisted, however
- It's best-practice to declare your variables at the top of the scope they are active in, even if you don't assign a value to them

```
console.log(a); // Raises exception
console.log(b); // does not
var b = 2;
```

# Global variables

- The global scope is actually an object, and global variables are properties of that object
  - In browsers, the global object is **window**
  - In Node.js, the global object is **global**

## Constants

- Read-only named identifiers
- As of ES5, there is no working implementation of constants
- ES6 provides the **const** keyword
  - Same naming and scope rules as variables
  - You cannot create a constant with the same name as a variable or function, and vice-versa

## Data types

- Five *primitive* data types:
    - **undefined**
    - **null**
    - **boolean**
    - **number**
    - **string**
- **object**

## Getting the type of a variable

- The **typeof** keyword, followed by a variable, will return the primitive type of that variable

```
var π = 3.14;
typeof π; // 'number'
```

# undefined & null

- Little difference between the two, practically
- Variables declared without a value will start as **`undefined`**
- Can compare to **`undefined`** to see if a variable has a value

```
var a;
a === undefined; // true
typeof a;        // undefined
```

## boolean

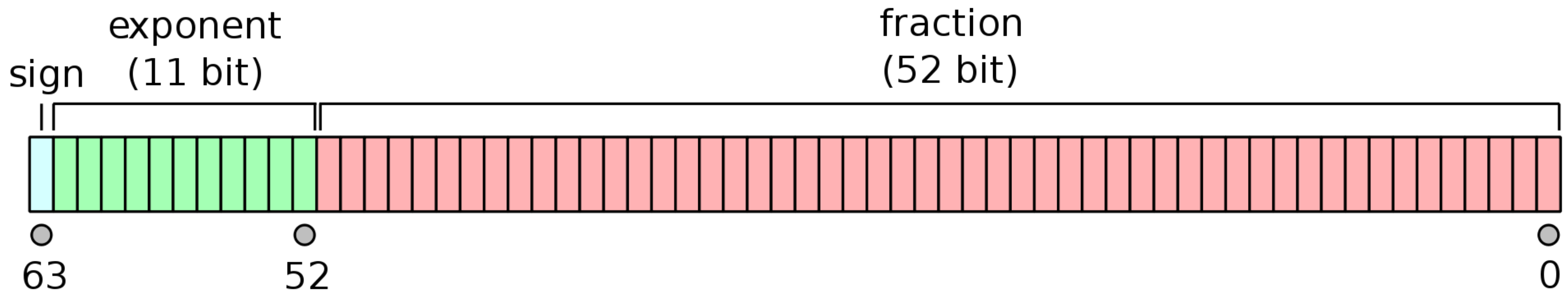
- **true** or **false**

## number

- Numbers can be expressed as:
  - Decimal: **-9.81**
  - Scientific: **`2.998e8`**
  - Hexadecimal: **`0xFF`**`; // 255`
  - Octal: **`0777`**
  - `Binary:` **`0b100000000000000000000000000000000`**
- Special numbers:
  - **`NaN`**
  - **`Infinity`**
  - **`-Infinity`**

## number gotchas

- All numbers are stored internally as 64bit floating points



- Integers are accurate up to 15 digits

```
var y = 9999999999999999; // 10000000000000000
```

- Decimals are accurate up to 17 digits

```
var x = 0.2 + 0.1;          // 0.30000000000000004
```

## string

- Enclosed by " or ' (just don't mix them)

```
var str = "My String";
```

- Escape with backslash (\)
  - **\n** is newline, **\t** is tab, etc
  - Works in both single- and double-quoted strings, but you shouldn't mix quote types
- + operator for concatenation

```
stringVar + " " + anotherStringVar + " !";
```

# Oh yeah, object!

- A list of zero or more pairs of keys and values, surrounded by curly braces
  - Would be considered a Dictionary, Hash or Map in other languages
- Keys are unordered, and must be a string
  - Keys need not be quoted, unless they contain characters that violate variable naming rules
- Values can be any type of data
- Keys and values can be created and retrieved with ease:

```
var obj = {bar: "baz"};
obj.foo = true;
obj.bar;    // "baz"
obj['foo']  // true
```

## Literals

- Fixed values, not variables, that you *literally* provide in your script

```
5          // number literal
"a"        // string literal
true       // boolean literal
{}         // object literal
[]         // array literal
/^(.*)$/  // regexp literal
```

## Literals, continued

- With the exception of **null** and **undefined**, every primitive type has a built-in object counterpart

```
    5 === Number("5")                  // true
```

- Literals are stored as low-level values, *until* they are accessed as if they were instances of their object counterpart!

```
    "I'm a string!".length             // 13
    "foo".toUpperCase()                // "FOO"
    0xFF.toString()                    // "255"
    {a:1}.hasOwnProperty('a')          // true
```

## Type Conversion

- If a variable type is not what JavaScript expects, it will convert it on the fly, based upon the context
- In expressions involving numeric and string values with the **+** operator, JavaScript converts numeric values to strings

```
x = "The answer is " + 42 // "The answer is 42"
y = 42 + " is the answer" // "42 is the answer"
```

## Implicit Conversion

- In statements involving other operators, it does not:

```
"37" - 7 // 30
"37" + 7 // "377"
8 * null // 0
```

- Hijinks ensue:

```
[] + []  // ?
[] + {}  // ?
{} + []  // ?
{} + {}  // ?
```

## Recap – Variables & Data Types

- There are **5** primitive types (**string, number, boolean, null, undefined**) and then **object**
  - Objects are collections of property names referencing data
  - Arrays are for sequential data
- Declare variables with **var**
- Types are implicitly converted, depending on context
  - Including when a primitive is used like an object
- *Almost* everything is an object, except the primitives
  - despite them having object counterparts

## Exercise – Variables & Data Types

- Declare some variables, get to know basic data types
- Fork this
  - http://jsfiddle.net/jmcneese/5rv25556

## Operators

- Unary
- Binary
  - Arithmetic
  - Relational
  - Equality
  - Bitwise
  - Logical
  - Assignment
- Ternary

## Unary

```
delete obj.x     // undefined
void 5 + 5       // undefined
typeof 5         // 'number'
+'5'             // 5
-x               // -5
~9               // -10
!true            // false
++x              // 6
x++              // 5
--x              // 4
x--              // 5
```

## Arithmetic

```
5 + 5          // 10
5 - 3          // 2
5 * 2          // 10
10 / 2         // 5
10 % 3         // 1
```

## Relational

```
'foo' in {foo: true}// true
[] instanceof Array // true
5 < 4               // false
5 > 4               // true
4 <= 4              // true
5 >= 10             // false
```

## Equality

```
5 == '5'          // true
5 !=   'a'        // true
5 === '5'         // false
{} !== {}         // true
```

# Bitwise

```
5 & 4   // 1
1 | 4   // 5
4 ^ 6   // 2
9 << 2  // 36
-9 >> 2// -3
9 >>> 2// 2
```

## Logical

```
false && 'foo'     // false
false || 'foo'     // 'foo'
```

## Assignment

```
x  =  5          //  5
x  +=  1         //  6
x  -=  2         //  4
x  *=  3         //  12
x  /=  4         //  3
x  %=  2         //  1
// &=,|=,^=,<<=,>>=,>>>=
```

## Ternary

```
true ? 'foo' : 'bar' // 'foo'
```

## Control Structures

- Conditionals
- Loops
- Flow Control
- Labels

## Conditionals

- if, else if, if
- switch

## if

```
var x = 5;
if (x === '5') {
      // foo
} else if (x >= 5) {
      // bar
} else {
      // baz
}
```

## switch

```
var x = 8;
switch (x) {
     case '8':
          // foo
          break;
     case 8:
          // bar
          break;
     default:
          // baz
}
```

## Falsy / Truthy

- These are **falsy**
  - `false`
  - `null`
  - `undefined`
  - `""`
  - `0`
  - `NaN`
- Everything else is **truthy**, including…
  - {}
  - []
  - "0"
  - "false"

## Loops

- for
- while
- do/while
- for … in

## for

```
for (var i = 0; i < 10; i++) {
     // do stuff 10 times
}
```

## while

```
var i = 0;

while (i < 10) {
    // do stuff 10 times
    i++;
}
```

## do/while

```
var i = 0;

do {
     // do stuff 10 times
     i++;
} while (i < 10);
```

## for...in

- Loop over *enumerable* properties of an object
  - Will include inherited properties as well, including stuff you probably don't want

```
var obj = {foo: true, bar: false};

for (var prop in obj) {
      prop;       // 'foo'
      obj[prop]; // true
}
```

## Flow control

```
for (var i = 0; i < 10; i++) {
      if (i < 5) {
            continue; // skip to next
      } else if (i === 8) {
            break;    // exit loop
      }
      console.log(i);
}
```

## Labels

- Labels can be applied to any block, and then passed to **break** or **continue**
  - Cannot be named a reserved word

## Labels continued

```
loop1: for (var i = 0; i < 3; i++) {
    loop2: for (var j = 0; j < 3; j++) {
        if (i == 1 && j == 1) {
            break loop1;
        }
        console.log(i, j);
    }
}
```

## Recap – Operators & Control Structures

- Conditionals
  - **if, else if, else** and **switch**
- Loops
  - **for, while, do/while,** and **for…in**
- Flow control
  - **break, continue** and labels
- Examples
  - http://jsfiddle.net/jmcneese/2bdt9s10

## Exercise – Operators & Control Structures

- Get to know control flow and iteration statements
- We'll use some basic browser functions

```
alert("A message!");
prompt("Ask for a value!");
confirm("Ask user to say 'ok'");
```

- Fork me
  - http://jsfiddle.net/jmcneese/hzchts18

## Exercise – FizzBuzz

- Let's put our new operator and control structure knowledge into practice!
- Write a program that console.logs the numbers from 1 to 100, separated by spaces
  - For numbers that are a multiple of 3, log "Fizz"
  - For numbers that are a multiple of 5, log "Buzz"
  - For numbers that are a multiple of both, log "FizzBuzz"
- Fork me
  - http://jsfiddle.net/jmcneese/tfx2ro28

## Functions

- Reusable, callable blocks of code
- Functions can be used as:
  - Object methods
  - Object constructors
  - Modules
- They are **First Class Objects**, and can have their own properties

## Defining a function

- Three ways
  - Function **declaration**
  - Function **expression**
  - with the `Function()` constructor
    - `new Function (arg1, arg2, … argn, functionBody)`
- http://jsfiddle.net/jmcneese/039gho4h

## Function declaration

- The function name is mandatory
- Cannot put a function statement in an if-block because of *hoisting*; it will become available to entire scope

```
function name(param, …) {
        // body
}
```

# Function expressions

- Define a function and assign it to a variable

```
var x = function <name>(param, …) {
     // body
};
```

- Name is optional in a function expression

```
// stores ref to anonymous function
var fn = function() {};
```

```
// stores ref to named function
var fn = function fn() {};
```

## Function invocation

- Invoke with `()`
  - `myFunctionName(argument1, argument2);`
- Any number of arguments can be passed in, regardless of defined parameters
- All arguments are available in **`arguments`** variable in the function

```
    function test() {
        arguments[0]; // first arg
    }
```

- Missing arguments will be set as **`undefined`**

## Function *pseudo*-parameters

- Every function has access to special variables within its body upon invocation
  - `arguments`
  - `this`

## arguments

- Available upon invocation
- Includes all arguments passed to the function
- Array-like, but not an actual array
  - Only has `length` property
- Mutable
  - Should be treated as read-only
- To treat like an array, convert it to one…

```
var arr = Array.prototype.slice.apply(arguments);
```

- http://jsfiddle.net/jmcneese/d1fy4rmm

## this

- Anyone have an idea what **this** is?

```
function fn() {
      return this;
}
```

## Return statements

- Functions do not automatically return anything, i.e. they are *void*
- To return the result of the function invocation, to the invoker (caller) of the function:

```
return <expression>;
```

- Constructor functions will automatically return **`this`**
- Careful with your line breaks…

```
return
      x;
// Becomes
return;
x;
```

## Global functions

- `isFinite`
- `isNaN`
- `parseInt, parseFloat`
- `encodeURI, encodeURIComponent`
- `decodeURI, decodeURIComponent`
- `setInterval, clearInterval`
- `setTimeout, clearTimeout`
- `eval            // dangerous`

## Exercise – Functional FizzBuzz

- Let's take the logic from our previous FizzBuzz exercise and make it functional
- Create a function that:
  - Accepts a single number argument
  - Returns the proper FizzBuzz result for that number
- Loop through 1…100 as before, but using the function to output the proper values
- Fork me
  - http://jsfiddle.net/jmcneese/6sxxzgpp

## Function Scope

- Scope refers to variable access and visibility in a piece of code at a given time
  - **var** declares a variable within the current scope
- Lexical/Static scope, not dynamic
  - Every inner level can access its outer levels
  - Scope is determined at definition
  - Refer to scope at two levels, *global* and *local*
- No block scope, instead JavaScript has function scope
  - Functions are the only thing that can create a new scope
  - A variable declared (with **var**) in a function is visible only in that function and its inner functions. But not the other way around.

## Function Scope continued

- Example of scope

```
var a = 5;
function foo(b) {
        var c = 10;
        d = 15; // no var; set in the global scope

        function bar(e) {
                var c = 2; // does not reference c
                a = 12; // will reference a
        }
}
```

- Each *inner* scope has access to the *outer*, **but** the outers cannot access the inners
- **ReferenceError** indicates unknown variable in scope
- http://jsfiddle.net/jmcneese/ekwnbaev

## Scope matters

- Avoid polluting the global scope
- You can "hide" things by wrapping them in a function
- Closures are born out of using lexical scope
- We'll see more of this later…

## No block scope…

- I lied…
  - eval() can cheat scoping, inserting itself into the scope
    - But… in *strict mode* eval() creates its own scope
  - An exception's "catch" block has its own scope for the exception argument variable

## Functions recap

- Functions
  - can be defined with a name or anonymously
  - are first class objects
  - create their own scope
  - Inner scopes can access parent scopes, but not the other way around
- Declare variables at the top of your scope to avoid hoisting issues

## Exercise – Function Scope

- Write two functions that share the same, global variable
  - Verify this by logging the value of the variable to the console from within each function
- Write two functions that share a non-global variable
  - Verify this by logging the value of the variable to the console from within each function
- Fork me
  - http://jsfiddle.net/jmcneese/78y7boxx

## Best Practices

- Avoid polluting the global namespace
- Define variables at top of your scope
- Use === & !== strict comparison
- Use named function expressions to avoid accidental hoisting
- Avoid primitive object wrappers like Number() or String()
- CamelCase constructor functions
- Include implicit ;
- Always open and close blocks with { }
- Indent and empty lines ensure readability

## Time for a pint!

- Q&A

## Day 2

- Custom and built-in objects
- Error handling
- JavaScript in the browser
- DOM API
- AJAX / XHR
- Final Lab

## Feedback?

- Any unanswered questions or lingering fear/uncertainty/doubt from yesterday?

## devdocs.io

- PLEASE use devdocs for upcoming exercises!

## Expositional vs. Professorial

- I will be showing and executing live examples of things as we go along
- Follow along in the console, trying out the things being shown/discussed
  - Try to replicate examples
  - Inspect variables in the console, to see what is available to tinker with
  - Don't be afraid to break stuff!

## PDF for today

- In case you didn't quite catch something, or can't see the screen well:
  - http://bit.ly/1LFRiy0

## Back to Objects…

- Remember that everything is an object except **`null`** and **`undefined`**
- Even primitive literals have object wrappers
  - They remain primitive until used as objects, for performance reasons
- An object is a dynamic collection of properties
  - Properties can be any type, including functions and objects
- **`this`** is a special keyword; inside an object method it refers to the object it resides in

## Object

- All built-in objects derive from the Object object
- Objects can be created with the **new** keyword
  - or via literal "object initializers", such as {}, [], /regex/
- Properties and methods on Object are inherited to all other Objects
- Some properties and methods only exist on the object constructor itself, these are called "generics" or "statics"
- Other properties and methods only exist on instances of the object in question

## Object – Generic Methods

- .create
- .defineProperty
- .defineProperties
- .getOwnPropertyDescriptor
- .getOwnPropertyNames
- .keys
- ... and more!

## Object.create

- Creates a new object with the specified prototype object and properties

```
var a = Object.create(Object.prototype, {
        foo: {
                writable: true,
                configurable: true,
                value: 'a'
        },
        bar: {
                writable: true,
                configurable: true,
                value: 'b'
        }
});
var a = {foo: 'a', bar: 'b'};
```

# Object.defineProperty

- Defines a new property directly on an object, or modifies an existing property on an object

```
Object.defineProperty(obj, 'x', {
      enumerable: true,
      configurable: true,
      value: 0,
      writable: true,
      get: function() {
            return this.x+10;
      },
      set: function(v) {
            this.x = v-10;
      }
});
```

## Object.defineProperties

- Defines new or modifies existing properties directly on an object, returning the object

```
Object.**defineProperties**(obj, {
        foo: {
                writable: true,
                value: 'a'
        },
        bar: {
                writable: true,
                value: 'b'
        }
});
```

## Object.getOwnPropertyDescriptor

- Returns a property descriptor for an own property

```
var obj = {x: 42};
Object.getOwnPropertyDescriptor(obj, 'x');
/* {
        configurable: true,
        enumerable: true,
        value: 42,
        writable: true
} */
```

## Object.getOwnPropertyNames

- Returns an array of all properties (enumerable or not) found directly upon a given object

```
var obj = {x: 42, y: 'a'};
Object.getOwnPropertyNames(obj);
// ['x', 'y']
```

## Object.keys

- Returns an array of a given object's own enumerable properties

```
var obj = {x: 42, y: 'a'};
Object.defineProperty(obj, 'z', {
    enumerable: false
});
Object.keys(obj); // ['x', 'y']
```

## Object Instance Properties

```
var n = new Number(5);
n.constructor === Number; // true
```

# Pass By Reference

- Objects are passed by reference, not value
  - So take care when passing an object into a function that modifies the object
- === operator compares object references, not values. Only true when objects are actually the same object in memory

## Mutability

- All primitives in JavaScript are immutable
  - Using an assignment operator just creates a new instance of the primitive
  - Pass-by-value
- Objects are mutable
  - Their values (properties) can change

## Exercise: Objects

- Create a "copy" function and method
- Fork me:
  - http://jsfiddle.net/jmcneese/ctwrkrzf

## Built-in Objects

- String
- Number
- Math
- Array
- Date
- RegExp
- Error
- http://jsfiddle.net/jmcneese/yxLugy6v

## String

- Instance properties

```
new String('foo').length // 3
```

- Instance method examples

```
var str = new String('hello world!');
str.charAt(0);         // 'h'
str.concat('!');       // 'hello world!!'
str.indexOf('w');      // 6
str.slice(0, 5);       // 'hello'
str.substr(6, 5);      // 'world'
str.toUpperCase();     // 'HELLO WORLD!'
```

## Number

- Generics

```
Number.MIN_VALUE
Number.MAX_VALUE
Number.NaN
Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINTY
```

- Instance method examples

```
var num = new Number(3.1415);
num.toExponential();     // "3.1415e+0"
num.toFixed();           // 3
num.toPrecision(3);      // 3.14
```

## Math

- Singleton-ish
- Methods
  - `abs, log, max, min, pow, sqrt, sin, floor, ceil, random…`
- Properties
  - `E, LN2, LOG2E, PI, SQRT2…`

## Array

- Generics

  Array.**isArray**([])            // true

- Examples
  - http://jsfiddle.net/jmcneese/qsxgvdnn

## Array mutator methods

```
var arr = new Array(1, 2, 3);
arr.pop();                // 3
arr.push(3);              // 3
arr.reverse();            // [3, 2, 1]
arr.shift();              // 3
arr.sort();               // [1, 2]
arr.splice(1, 0, 1.5);  // [1, 1.5, 2]
arr.unshift(0);           // [0, 1, 1.5, 2]
```

## Array accessor methods

```
var arr = new Array(1, 1);
arr.concat([2, 4]);       // [1, 1, 2, 4]
arr.join('-');            // "1-1"
arr.slice(1, 1);          // [1]
arr.toString();           // "1,1"
arr.indexOf(2);           // -1
arr.lastIndexOf(1);       // 1
```

# Array iteration methods

```
var arr = new Array(1, 1, 2, 4);
arr.forEach(fn);
arr.every(fn);
arr.some(fn);
arr.filter(fn);
arr.map(fn);
arr.reduce(fn);
arr.reduceRight(fn);
```

## Array – Exercise

- Let's work with some arrays
- Fork me:
  - http://jsfiddle.net/jmcneese/c5g998fd

## Date

- Represents a single moment in time based on the number of milliseconds since 1 January, 1970 UTC

```
    new Date();
    new Date(value);
    new Date(dateString);
    new Date(year, month[, day[, hour[, minutes[, seconds[, milliseconds]]]]]);
```

- Examples
  - http://jsfiddle.net/jmcneese/76aat2kc

## Date Methods

- Generics

```
Date.now()
Date.parse('2015-01-01')
Date.UTC(2015, 0, 1)
```

- Instance method examples

```
var d = new Date();
d.getFullYear();            // 2015
d.getMonth();               // 7
d.getDate();                // 15
```

## Date – Exercise

- Do you want to exercise this?

# RegExp

- Creates a regular expression object for matching text with a pattern

```
var re = new RegExp("\w+", "g");
var re = /\w+/g;
```

- Generics

```
var re = new RegExp("\w+", "g");
re.global;          // true
re.ignoreCase;      // false
re.multiline;       // false
re.source;          // "\w+"
```

- Examples
  - http://jsfiddle.net/jmcneese/8jnso5wf

# RegExp Methods

- Instance methods
  - re.**exec**(*str*)
  - re.**test**(*str*)
- String methods that accept RegExp params
  - str.**match**(*regexp*)
  - str.**replace**(*regexp, replacement*)
  - str.**search**(*regexp*)
  - str.**split**(*regexp, limit*)

## Function

- Creates a new Function object
  - Every function is actually a Function object
- Generics

```
function foo(x,y,z) {}
foo.length;          // 3
```

- Examples
  - http://jsfiddle.net/jmcneese/tqbdzpc1

# Error

- Error objects are thrown when runtime errors occur
  - Can also be used as a base objects for user-defined exceptions

```
var err = new Error('Oh noes!');
```

- Implementation varies across vendors
- Instance properties

```
err.name;        // "Error"
err.message;     // "Oh noes!"
```

## Error Handling

- JavaScript is very lenient when it comes to handling errors
- Internal errors are raised via the **throw** keyword, and are then considered "exceptions"
- Exceptions are handled via a **try/catch/finally** construct, where the thrown exception is passed to the **catch** block
  - Nesting allowed
  - Exceptions can be re-thrown
- *Anything* can be thrown, of any data type
- Uncaught exceptions halt the overall script
- Example
  - http://jsfiddle.net/jmcneese/m83pgvbn

## Error Handling – Exercise

- If we have time and/or desire, we can practice throwing errors and handling them...

## Strict Mode

- `"use strict";`
- It kills deprecated and unsafe features
- It changes "silent errors" into thrown exceptions
- It disables features that are confusing or poorly thought out
    - Ensures "eval" has its own scope
    - Does not auto-declare variables at the global level during a scope-chain lookup
- Can be set globally or within function block
- http://jsfiddle.net/jmcneese/kuj83fyL

## JavaScript and the Browser

- Why JavaScript
  - Interactivity, based on user or browser triggered events
  - Load data into the page dynamically
  - Built in business logic
  - Single page applications (if that's your cup of tea)
  - The web is the new application platform…
- How it fits
  - HTML for view data & ui structure
  - CSS for presentation
  - JavaScript for behavior
    - Though.. becoming central for business logic

## Skill check

- Write an HTML form from scratch?
- Style a form (or full page) from scratch?
- Manipulate elements in the page with just the DOM?
- Set up an event handler for form submissions? Clicks?
- Know what events are and why they are important?

## Browser debugging w/ Console

- "`console`" object
  - Typically on "`window`", though doesn't always exist
  - Methods
    - `log`
    - `dir (lists all properties)`
    - `info`
    - `warn`
    - `error`
    - `table(object)`
    - `group(name); groupEnd();`
    - `assert(expr, message); // shows only if false`
- http://jsfiddle.net/jmcneese/yuhpphsx

## HTML

- **H**yper**T**ext **M**arkup **L**anguage
- Browsers allow support for all sorts of errors – html is very error tolerant
- Structure of the UI and "view data"
- Tree of element nodes
- HTML5
  - Rich feature set
  - Semantic
  - Cross-device compatibility
  - Easier!

## Anatomy of a page

```
<!doctype html>
<html lang="en">
     <head>
          <meta charset="utf-8">
          …document info and includes…
     </head>
     <body>
          <h1>Hello World!</h1>
     </body>
</html>
```

## Anatomy of an element

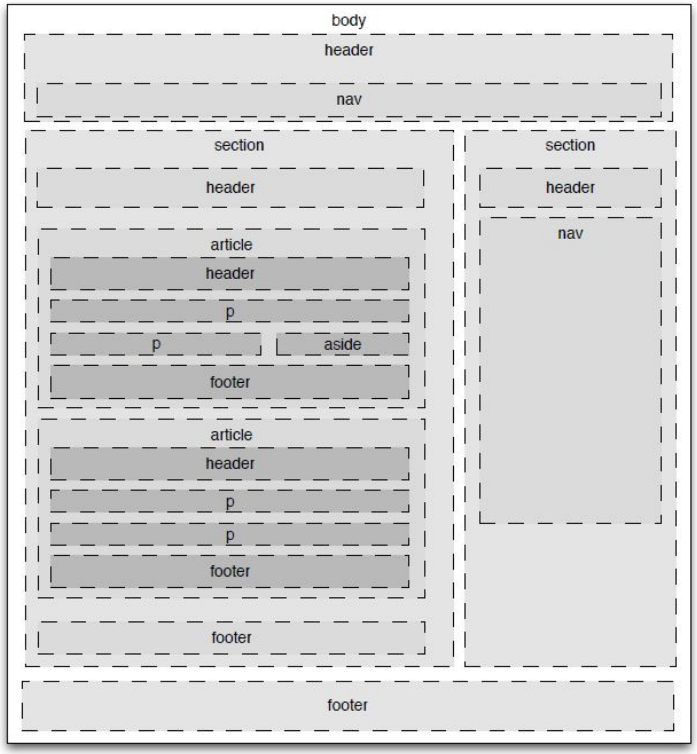
- <element attributeName="attributeValue">
  *Content of element*
  </element>
- Block vs inline
  - <p></p>
  - <strong></strong>
- Self closing elements
  - <input type="text" name="username" />

## HTML Elements refresher

- Structure
  - <div>
  - <span>
  - <table>
    - <tr>, <td>, <thead>, <tbody>
  - <form>
    - <fieldset>, <label>, <input>, <select>, <textarea>
  - And some new HTML5 semantic elements
- Content
  - <h1> through <h6>
  - <p>
  - <ol> or <ul> (with<li>)
  - Text modifiers
    - <em>, <strong>
- See:
  - https://developer.mozilla.org/en-US/docs/Web/HTML/Element

# HTML5 Semantic Elements

- Designed to degrade gracefully on non-HTML5 browsers
- Define an outline and semantic hints for a document
  - <header>
  - <footer>
  - <nav>
  - <main>
  - <section>
  - <article>
  - <aside>
  - <figure>, <figcaption>
  - <time>
  - <mark>
  - <details>, <summary>
- http://jsfiddle.net/jmcneese/ztzphgkh/



# HTML5 Forms

- New input types
  - number
  - range
  - url
  - email
  - tel
  - color
  - Search
- New element
  - datalist
- New input attributes
  - required
  - autofocus
  - placeholder
  - List
- Built in validation
  - Use "novalidate" to turn off
- http://jsfiddle.net/mrmorris/zh18vn4x/

## CSS

- **C**ascading **S**tyle **S**heets
- Language for describing look and formatting of the document
- Separates presentation from content
- Define as external resource or inline
  - `<link rel="stylesheet" type="text/css" href="theme.css">`

  -
    ```
    <style type="text/css">
     …style declarations…
    </style>
    ```
  - `<span style="color:#FF0000">RED</span>`

## Anatomy of a css declaration

-
  ```
  selector(s) {
      /* declaration block */
      property: value;
      property: value;
      property: val1 val2 val3 val4;
  }
  ```

-
  ```
  div {
      color:#f90;
      border:1px solid #000;
      padding:10px;
      margin:5px 10px 3px 2px;
  }
  ```

## CSS Selectors

- By element
  - h1 {color:#f90;}  <h1></h1>
- By id
  - #header {}  <div id="header"></div>
- By class
  - .main {}  <div class="main"></div>
- By attribute
  - div[name="user"] {}  <div name="user"></div>
- By relationship to other elements
  - li:nth-child(2) {}  <ul><li></li>***<li></li>***</ul>

# CSS Specificity

- Multiple selectors will override each other based on "specificity"

  -
    ```
    <div id="main" class="fancy">
     What color will I be?
    </div>
    ```

  -
    ```
    .fancy{
        color: blue;
    }
    #main{
     color: green;
    }
    #main.fancy{
     color: red;
    }
    ```
- Order of specificity: inline, id, psuedo-classes, attributes, class, type, universal
- !important

## New Selectors & Classes

- Attribute Selectors
  - http://jsfiddle.net/jmcneese/nyp920L0
- Sibling Selector
  - http://jsfiddle.net/jmcneese/2fLz686s
- Pseudo-classes
  - Form inputs
    - http://jsfiddle.net/jmcneese/9bke447m
  - Structural
    - nth/first/last/only
    - :empty
    - :not
    - http://jsfiddle.net/jmcneese/3qkmzt48

# Including JS on the page

- HTML Parser parses while script is downloaded then run
- Include at the bottom of `</body>`
  - It won't block
  - All DOM is loaded
- `<script>…</script>` blocks
- `<script src="filename.js"></script>`
- Inline on elements is possible `"onclick"`

## How JavaScript loads

- JS loading is blocking…
- Browser will download then interpret any JS it comes across before proceeding
- So… put scripts in files and at bottom of the body tag

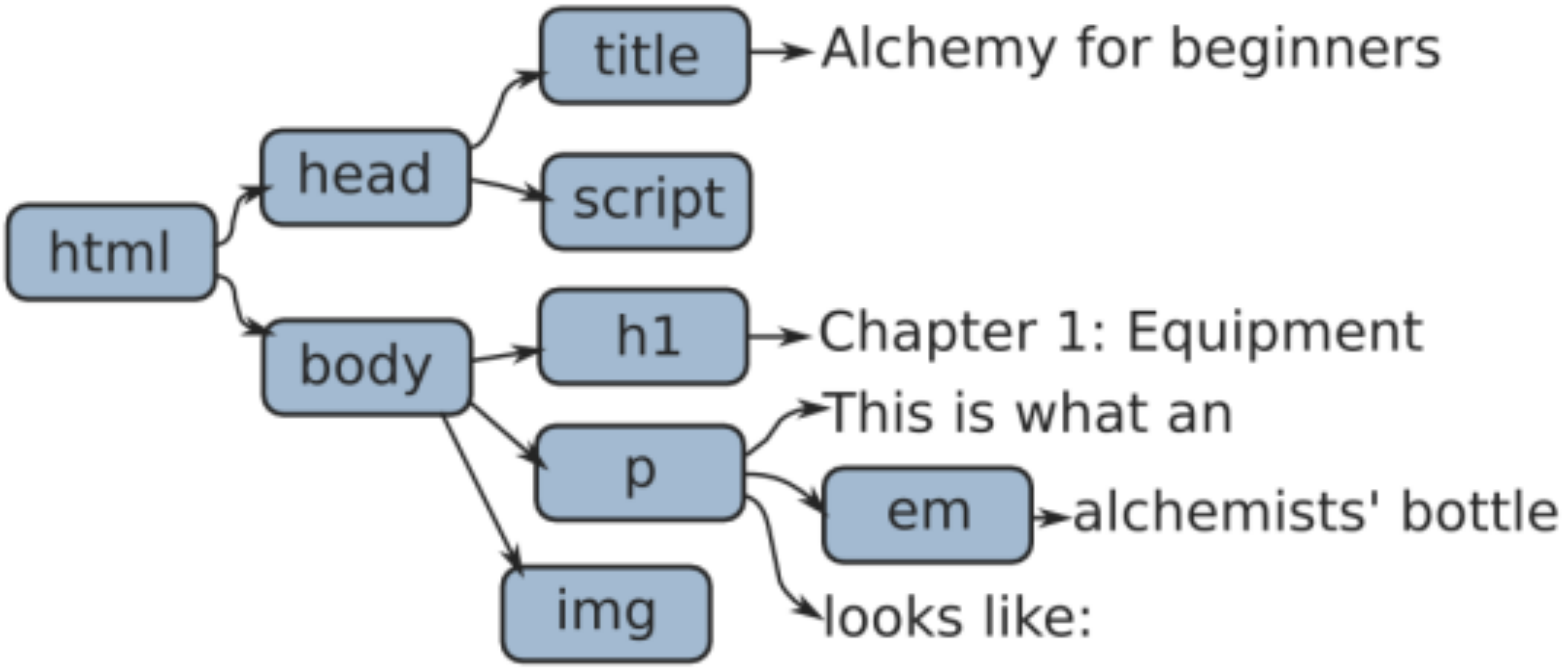
## The DOM

- **D**ocument **O**bject **M**odel
- What most people hate when they say they hate JavaScript
- The browser's API: interface it provides to JavaScript for manipulating the page
- Browser parses HTML and builds a model of the structure, then uses the model to draw it on the screen
- "Live" data structure

## Document structure

- Global `document` variable gives us programmatic access to the DOM
- It's a tree-like structure
- Each node represents an **element** in the page, or **attribute**, or **content** of an element
- Relationships between nodes allow traversal
- Each DOM node has a **nodeType** property, which contains a code for the type of element…
  - 1 – regular element
  - 3 – text

## Document Structure



## Document Nodes

- <p id="name" class="hi">My text</p>
- Maps to: {

```
    …
    childNodes: NodeList[1],
    className: "hi",
    innherHTML: "My text",
    id: "name",
    …
}
```

- Attributes map **very loosely** to object properties so don't rely on it

# Working with the DOM

- Access the element(s)
  - Select one
  - Select many
  - Traverse
- Work with the element(s)
  - Text
  - Html
  - Attributes

# DOM - Traversing

- Move between nodes via their relationships
- Element node relationship properties
  - `.parentNode`
  - `.previousSibling, .nextSibling`
  - `.firstChild, .lastChild`
  - `.childNodes // NodeList`
- Mind the whitespace!
- http://jsfiddle.net/jmcneese/ycf19k7L

## DOM – Get your element(s)

- Start on **`document`** or a previously selected element
- Returns a `NodeList`
  - `.`**`getElementsByTagName`**`("a");`
  - `.`**`getElementsByClassName`**`("className");`
  - `.`**`querySelectorAll`**`("css selector");`
- Returns a single element
  - `document.`**`getElementById`**`("idname");`
  - `.`**`querySelector`**`("css selector");`
- http://jsfiddle.net/jmcneese/n9mvhjf7

## Looping through a NodeList

- **`HTMLCollectionObject/NodeList`**
  - An array-like object containing a collection of DOM elements
  - The query is re-run each time the object is accessed, including the **length** property

## DOM – Node Content

- Text node content
  - `textNode.`**`nodeValue`**
- Element node content
  - `el.`**`textContent`**
  - `el.`**`innerText`**
  - `el.`**`innerHTML`**

## DOM – Manipulation

- http://jsfiddle.net/jmcneese/ep9j6qun
- Use `el.innerHTML`
- DOM manipulation
  - `.createElement("div")`
  - `.createTextNode("foo bar")`
  - `.appendChild(el)`
  - `.removeChild(el)`
  - `.insertBefore(newEl, beforeEl);`
  - `.replaceChild(newEl, oldEl)`

## DOM – Element Attributes

- http://jsfiddle.net/jmcneese/qhssoxey
- Accessor methods
  - `el.getAttribute(name);`
  - `el.setAttribute(name, value);`
  - `el.hasAttribute(name);`
  - `el.removeAttribute(name);`
- As properties
  - `.href`
  - `.className`
  - `.id`
  - `.checked`

# DOM – Styling/CSS

- Use element's **style** property
  - It's an object of style properties
    - **.color** = "black";
    - **.position**
    - **.top**
    - **.left**
- Some style names differ in JavaScript
  - Hyphens become camelCase
    - background-color => backgroundColor
  - Some names were keywords
    - float => cssFloat
- http://jsfiddle.net/jmcneese/fezqxjdz

## DOM - Getting styles

- Can't accurately "get" styles on an element through its style property or attribute
- Must use `window.`**`getComputedStyle`**`()`
- `el.`**`cssText`** will return computed inline styles
- classList API
  - `el.classList.`**`add`**`("class");`
  - `el.classList.`**`remove`**`("class");`
  - `el.classList.`**`toggle`**`("class");`
  - `el.classList.`**`contains`**`("class")`

# Geometry of Elements

- Element size in px
  - `.offsetWidth`
  - `.offsetHeight`
- Element inner size, ignoring borders
  - `.clientWidth`
  - `.clientHeight`
- **`el.getBoundingClientRect()`**
  - Returns object with top, bottom, left, right properties relative to top left of the page

# DOM Performance

- Dealing with the DOM brings up a lot of performance issues
- Touching a node has a cost (especially in IE)
- Styling a bigger cost (it cascades)
  - Inserting nodes
  - Layout changes
  - Accessing css margins
  - Reflow
  - Repaint
- Accessing a **`nodeList`** has a cost

## DOM basics - Recap

- The **DOM** is a model of the web page document.
  - It is a standardized convention.
- Browsers offer a **JavaScript API** to interact with the DOM
  - Can affect the page
- You can access, manipulate, create any content within the page
- jQuery will abstract much of the DOM API implementation nuances away, but it is still a good set of tools to have under your belt
  - **document.getElementById()**
  - **el.querySelector()**
  - **el.querySelectorAll()**

## Exercises: Dom manipulation

- Using your special DOM hunting and walking skills, find the 3 "FLAG" elements in the content and move them to the "#bucket" element
  - http://jsfiddle.net/jmcneese/Lenn7h3g
- Make a function, "embolden", that takes an element and makes it bold
  - 
```
function embolden(element) {
 // makes the element bold
  // hint: style it OR wrap it in <strong>
}
```

## Events

- JavaScript engine has an event-driven, single-threaded, asynchronous programming model
- As things happen
  - User clicks
  - Page completes loading
  - Form is submitted
- Events are fired
  - Click
  - Load
  - Submit
- Which can trigger handler functions that are listening for these events

## So many events…

- UI (load, unload, error, resize, scroll)
- Keyboard (keydown, keyup, keypress)
- Mouse (click, dblclick, mousedown, mousemove mouseup mouseover, mouseout)
- Focus (focus, blur)
- Form (input, change, submit, reset, select, cut, copy, paste)

## Event handling (Basics)

- Select an element that relates to the event
- Indicate which event you want to listen for
- Define an event handling function to respond to the event when it occurs

# Event handling evolution

- Netscape
  - Used on<type>
  - Node["onClick"] = function(){..}
- Microsoft
  - Node.attachEvent
  - Has global event object
- W3C
  - Node.addEventListener
    - All browsers by ie9+
  - body.addEventListener("click", function(){});

## Bind an event to an element

- Inline
- Traditional DOM event handlers
  - `el.onclick = function(){}`
- Event listeners (ie9+)
  - `el.addEventListener(event, function [, flow]);`
  - `el.removeEventListener(event, function);`
  - `el.attachEvent(); // ie8- only`
- Handlers are passed an "`event`" object
  - event object can have different properties depending on the event (ex: "`which`" for key pressed)
- http://jsfiddle.net/jmcneese/xwts6wuv

## Context… in event handling

- JavaScript event handlers will be invoked in the context of the DOM node that triggered the event
  - So: this === elementNode;
- Careful with inner functions or callbacks, which may change context
  - Can store parent context
  - Or use fn**.bind**

## Event handlers with arguments

- This won't work like you expect

```
element.addEventListener('blur', doSomething(5));
```

- It is passing the result of a function invocation, not the function to-be-called as a callback
- Instead, wrap it in a function

```
el.addEventListener('blur', function() {
        doSomething(5); // func needed an arg
});
```

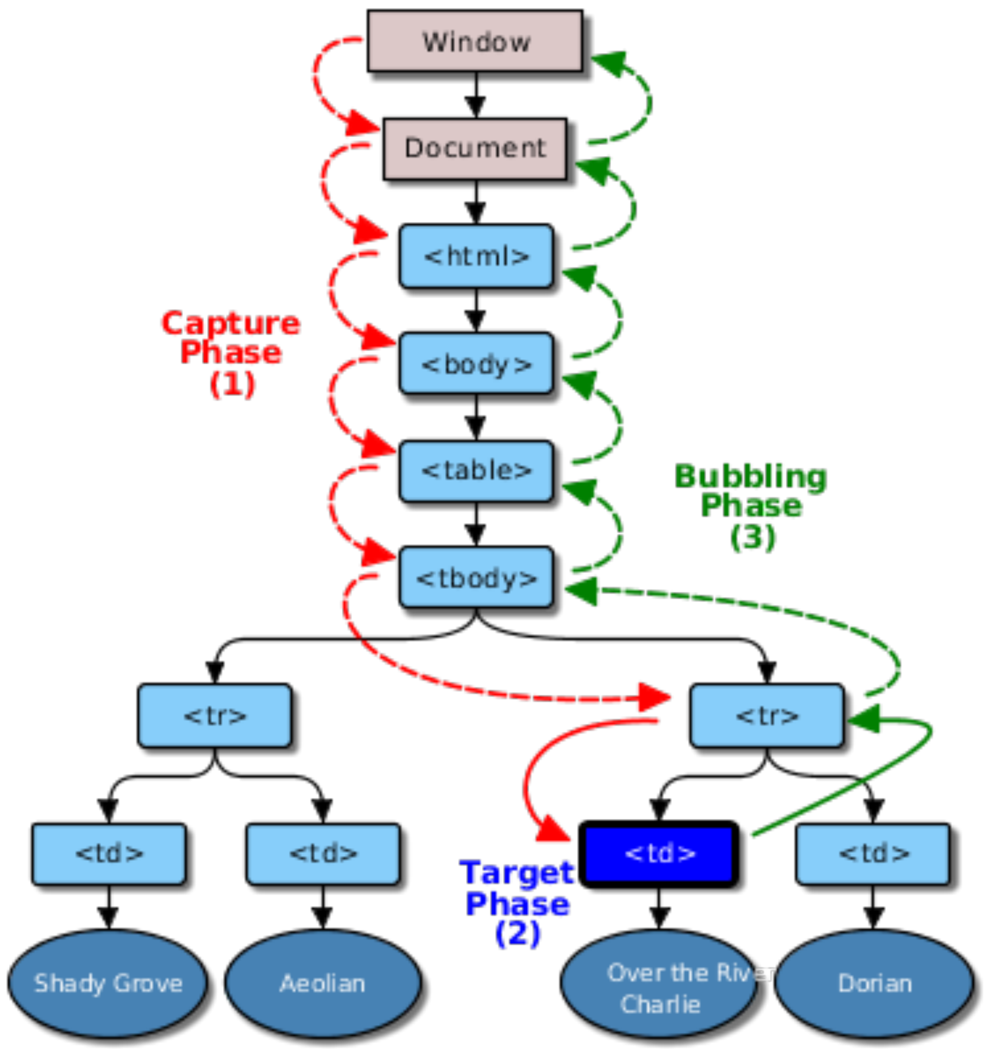
- Or use Function.**bind**

```
el.addEventListener(
        'blur',
        doSomething.bind(el, 5)
);
```

# Event Propagation

- An event triggered on an element is also triggered on all parent elements of the element
- Two models
  - Trickling, aka Capturing phase (Netscape)
  - Bubbling (MS)
- W3C decided to support both
  - Starts in capturing, then bubbling
  - Defaults to bubbling
- Bubbling supports Event Delegation
  - attach an event handler to a common parent of many nodes… and parent can determine source child and dispatch as needed.

# Event Propagation, continued



# Stopping events

- Cancel bubbling

  -
    ```
    // ie8 and below
    eventObject.cancelBubble = true;

    // ie9+
    if (eventObject.stopPropagation) {
     eventObject.stopPropagation();
    }
    ```
- Prevent default browser behavior
  - `eventObject.preventDefault()`

## Event Delegation

- When a parent element is responsible for handling an event that bubbles up from its children
  - Allows new child content to be added and support the same event
  - Fewer handlers registered, fewer callbacks, reduced chance for memory leaks
- Relies on some event object properties
  - `target`, which references the originating node of the event
  - `currentTarget` property refers to the element currently handling the event (where the handler is registered)
- http://jsfiddle.net/jmcneese/8wb1wrkp

## Event Metadata

- Can determine the location of the mouse when the event occurred
  - Event**.screenX**
  - Event**.screenY**
  - Event**.pageX**
  - Event**.pageY**
  - Event**.clientX**
  - Event**.clientY**
- Key events include a **keyCode** property
- http://jsfiddle.net/jmcneese/1vqmbbjw

## Event de-bouncing

- "debounce" events that fire rapidly like **mousemove**, **scroll, key***

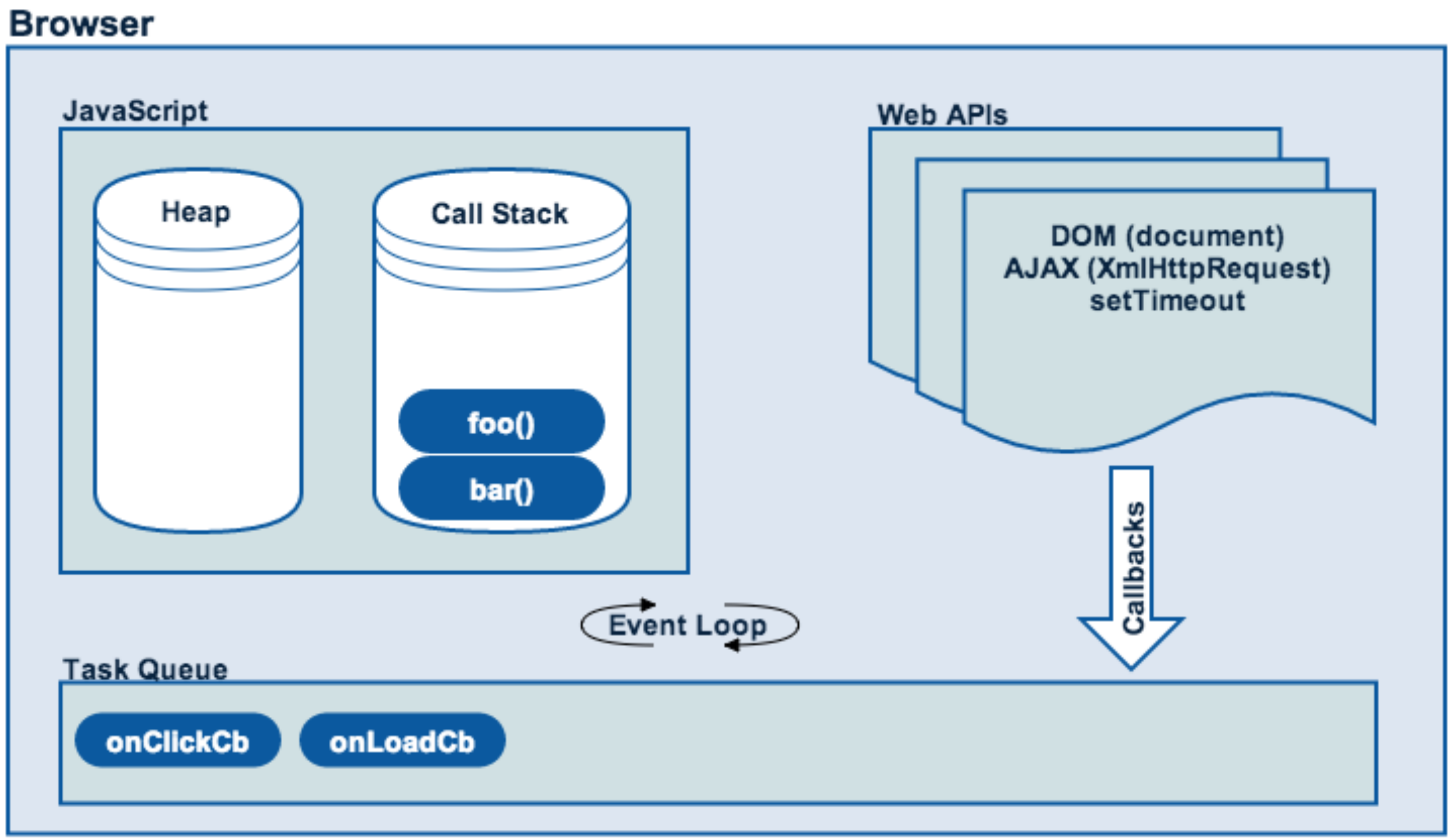
```
textarea.addEventListener("keydown", function(){
        clearTimeout(timeout); // safe
        timeout = setTimeout(function(){
                // stopped typing…
        }, 500);
});
```

- Respond to an event at intervals, displaying mouse coordinates periodically on **mousemove**
- http://jsfiddle.net/jmcneese/k7wr8o6x

## Event Loop

- The true power of JavaScript
- Allows asynchronous operations
  - Methods that get tacked into the queue are 'async'
- Each tick it returns a function from the Queue and runs it to completion (blocking)
  - Avoid blocking scripts…
    - alert, confirm, prompt, synchronous XHR
- For long running tasks…
  - *Eteration*: break task into multiple turns and call each with a setTimeout
  - WebWorkers: Move the task into a separate process

# The event loop



## The event loop break down

- Single threaded stack
- Function calls are added to the stack
- On stack… "return" will pop the call off the stack and go to the next function down
- Each call is blocking until finished
- When stack is clear, anything (next one) in the Task Queue is popped off and put on the stack to run

## Events recap

- **Events** are notifications that bubble up from different sources in the page
  - User action (click, type, scroll)
  - Browser action (load, unload)
- Event delegation allows you to register a single handler to handle many (child) nodes' events
- The event loop is powerful but it is single-threaded so a long-running process can halt all interactions in the page

## Events – Exercise

- Our final exercise today will include user events, and be somewhat duplicative of any exercise we do here
  - But... we can still practice it, if you want/need to

## AJAX / XHR

- Interface through which browsers can make HTTP Requests
- Handled by the **`XMLHttpRequest`** object
- Introduced by Microsoft in the 90s for IE, for Outlook Web Access
- Why use it?
  - Non-blocking
  - Dynamic page content/interaction
  - Supports many formats
- Limitations
  - Same-origin policy
  - History management

## XHR – Sending a request

- Step by step
  - Browser makes a request
  - Server responds
  - Browser processes response
- Create a request object and call its **.open** and **.send** methods

```
var req = new XMLHttpRequest();
req.open("GET", "url.json");
req.send();
```

## XHR – Handle the response

- *load* event will fire when response is received

-
  ```
  req.onload = function(e) {
      if (req.status==200) {
          console.log(req.responseText);
      }
  }
  ```

-
  ```
  req.addEventListener("load", function(e) {
      // also ok
  });
  ```

## XHR – Example

- http://jsfiddle.net/jmcneese/8dkoadfw

# Ajax Data formats

| Format | Summary | PROS | CONS |
|---|---|---|---|
| HTML | Easiest for content in page | • Easy to parse<br>• No need to process much | • Server must produce the HTML<br>• Data portability is limited<br>• Limited to same domain |
| XML | Looks similar to HTML, more strict | • Flexible and can handle complex structure<br>• Processed using the DOM | • Very verbose, lots of data<br>• Lots of code needed to process result<br>• Same domain only |
| JSON | Similar object literal syntax | • concise! Small<br>• Easy to use within JavaScript<br>• Any domain, w/ JSONP or CORS | • Syntax is strict<br>• Can contain malicious content since it can be parsed as JavaScript |

## XHR with HTML

- Easiest way to go
- Works with the DOM and styles
- Scripts will NOT run

## A word about JSON

- **J**ava**S**cript **O**bject **N**otation
- Most commonly used web data communication format
- Methods
  - `JSON.stringify(object);`
  - `JSON.parse(string);`

# JSON

```
{
        "name": "Jason",
        "trophies": [
                "trophy1",
                "trophy2"
        ],
        "age": 40,
        "car": {
                "name": "toyota",
                "year": 1985
        }
}
```

## XHR with JSON

- It is sent and received as a string and will need to be deserialized

```
	var data = JSON.parse(xhr.responseText);
	var newContent = "";

	data.forEach(function(img) {
		newContent += '<img src="'+img.src+'"/>';
	});
	document.getElementById('images').innerHTML = 
newContent;
```

# XHR continued

- It is best to use an abstraction of `XMLHttpRequest` for
  - **`status`** and **`statusCode`** handling
  - Error handling
  - Callback registration (**`onreadystatechange`** vs **`onload`**)
  - Browser variations and fallbacks
  - Additional event handling
    - `progress`
    - `load`
    - `error`
    - `abort`
- Use a lib like jQuery….

## Cross-origin restrictions

- By default, XHR requests must be made on the same domain as the originating script, because of their ability to perform advanced requests (POST, PUT, DELETE), as well as specify custom HTTP headers
- How to get around this restriction, if you want to consume data from a non-origin server?

# JSONP

- Browsers don't enforce same-origin policy on the `src` in script tags
- Shenanigans:
  - We define a handling **callback** function
  - We dynamically add a **script** referencing an external `script`
  - We tell the script the **callback** to wrap the response in
  - Once script loads, the response is wrapped in our **callback**, which is invoked on load
- Caveats
  - Only works with GET requests
  - Does **not** use XMLHttpRequest
  - Super insecure and shouldn't ever be used in conjunction with untrusted third parties due to CSRF, XSS and other exploits

## CORS

- Cross-Origin Resource Sharing
- A set of headers sent by the requesting client (XHR) and the responding server that can negotiate whom can request what from where
- Caveats
    - Supports **all** HTTP verbs
    - Usable with XMLHttpRequest
    - Simple in theory, complex in practice

## XHR Recap

- A means for the browser to make additional requests without reloading the page
- Enables very fast and dynamic web pages
- Best with small, light transactions
- **JSON** is the data format of choice
- Requests across domains are possible but require jumping through some extra hoops (and your server must support it)

## Final Lab

- Normal – Create a tabbed UI that responds to click events and adding new tabs.
  - http://jsfiddle.net/jmcneese/25no442s
- Hard Mode – Either:
  - Create a table-builder function that requests JSON data and builds a table element with all the trimmings.
    - http://jsfiddle.net/jmcneese/esbu0nmd
  - Modify the weather example to have a way for the user to specify a postal code for weather conditions
    - Original fiddle: http://jsfiddle.net/jmcneese/8dkoadfw
    - YQL Weather API: https://developer.yahoo.com/weather

## That's all, folks!

- Q&A