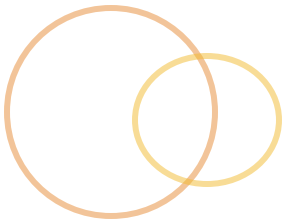
Core JavaScript

Ryan Morris
Develop Intelligence

Introductions

- Who am I?
- Who are you?
  - What do you do?
  - What do you hope to gain from this course?
  - What is your programming background?
    - Any JavaScript, HTML, CSS, jQuery, etc?
    - Are you coming in with love or hate for js?
- What is your current development environment and process?

How the class works

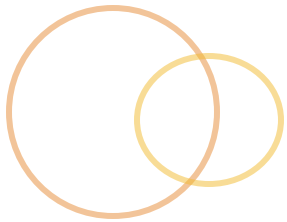


- Lecture mixed with labs
- Informal
  - Stop me anytime for questions or more information
  - Daily outline is flexible, we'll adjust as needed
    - Useless topic? Tell me!
    - Topic to add to the syllabus? That's ok, too!
- Timing
- Class review at the end

Getting the most out of the class

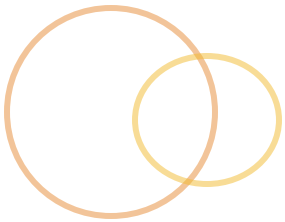
- Ask questions!
- Do the labs (pair up if needed)
- Be punctual
  - Class starts with or without you
  - If you can't make it then let me know
- Avoid meetings and work distractions…
  - if you can
- Master your google-fu
- Play along in the console
- Don't be afraid to break stuff

Course overview



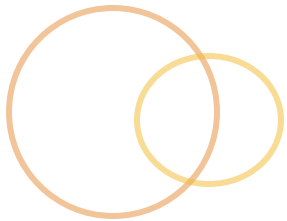
- Day 1: JavaScript Basics
- Day 2: HTML and the DOM, Events, Ajax

Goals for this class



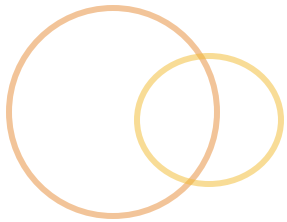
- Become familiar with JavaScript's:
  - Lexical structure
  - Data types
  - Operators
  - Control structures
  - Functions
  - Objects
- Learn about:
  - JavaScript in the browser
  - DOM API
  - AJAX / XHR
- Be able to:
  - Build a webpage that uses JavaScript to respond to user interaction, and dynamically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

Resources



- Documentation
  - http://devdocs.io
  - https://developer.mozilla.org/en-US/docs/Web
  - http://kapeli.com/dash (Mac only)
  - Google it.
- Compatibility checks
  - http://caniuse.com
- ES 5 compatibility table
  - http://kangax.github.io/compat-table/es5/

Set up for exercises

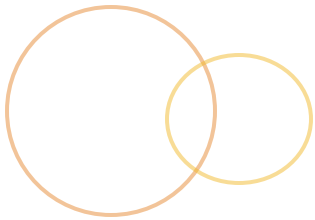
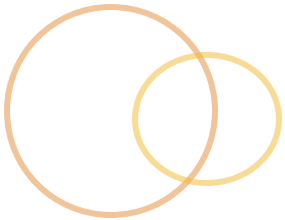


- A browser with dev tools
  - Preference for Chrome in class
  - Open your browser and hit F12
- Exercise Application
  - https://github.com/rm-training/core-javascript
  - Download the lastest .zip of *master*
- A text editor or IDE
- Our web editor, jsfiddle
  - http://jsfiddle.net/

Need this PDF?

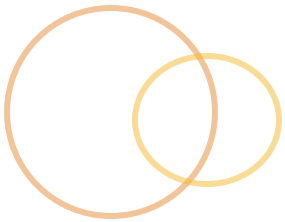
- <a link>

Day 1

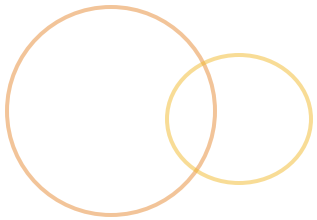

- JavaScript History & Today
- Core data types, syntax, control flow
- Data structures
- Functions, scope, closures
- Object inheritance (-lite)

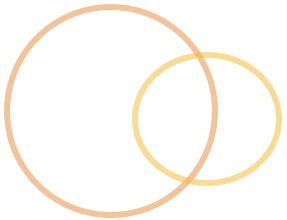
What is JavaScript?



- Standardized as **ECMAScript**
- **Interpreted**
- Case-sensitive C-style syntax
- Dynamically typed (with weak typing)
- Fully **dynamic**
- **Single-threaded** event loop
- Unicode (UTF-16, to be exact)
- **Prototype**-based (vs. class-based)
- Kind of weird but enjoyable

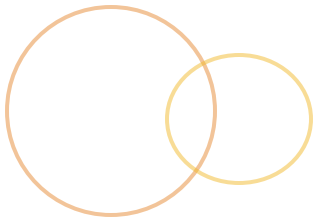
History

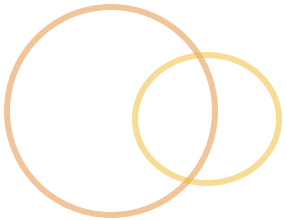




- 1995 - Netscape wanted interactivity like HyperCard w/ Java in the name
- Designed & built in 10 days by Brendan Eich as "Mocha", released as "LiveScript"
- Combines influences from:
  - Java, "Because people like it"
  - SmallTalk, prototypal

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 services
  - Beginning to dominate the entire software stack
- Easy to learn, hard to master

Approaching JavaScript

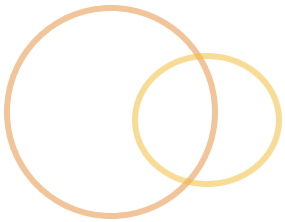
- It's *not* Java
- It's *not* class-based
- Very dynamic
- Supports imperative, functional and object-oriented approaches

Obligatory Hello World

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

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

Browser Console



- Use browser dev tools to access its JavaScript console
  - The browser's "`console`" is a line interpreter and log
- All major browsers are converging to the same API for console debugging
- 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() => echo`

Module: Language fundamentals

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

Fundamentals examples

- http://jsfiddle.net/mrmorris/23zK2/

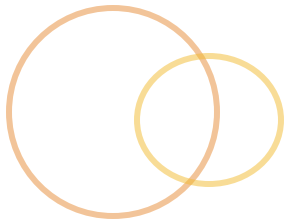
C-family syntax

- Instructions are **statements** separated by **semi-colon**
- Spaces, tabs and newlines are **whitespace**.
- White space and indentation generally doesn't matter
- **Blocks** are wrapped with curly braces { }

Automatic Semicolon Insertion

- Semicolons terminate statements
  - var x = 1 + 2;
- Are sorta-optional
  - JavaScript will automatically insert them where it needs them, but not fail-safe
  - But… don't rely on it…

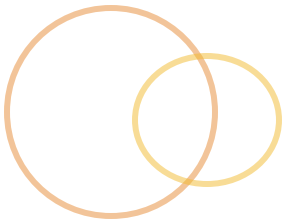
Comments



- 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
 */
```

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];
```

- ES6: supports **let** keyword

```
let x = 1; // difference in scope
```

Variable names

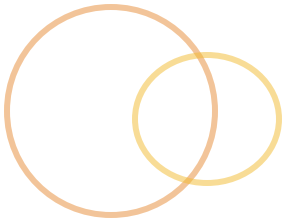
- Must start with a letter, underscore (_), or dollar sign ($)
- Subsequent characters can be alphanumeric, _ or $
- Case-sensitive!
- Unicode characters are supported

```
var 🍔 = 'burger';
```

Variable scope

- **Function scope** (not block scope)
  - Variables declared outside of a function, or by omitting the **var** keyword within a function, are "**global**"
  - Variables created within a function are called "**local**" or "**function**" variables
- ES6 supports block-scope with ***let***
  - if (expression) {
    let x=1; // scoped to this block only
    }
    console.log(let); // Temporal Dead Zone error

Data types



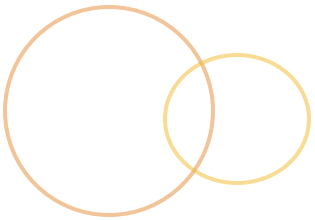
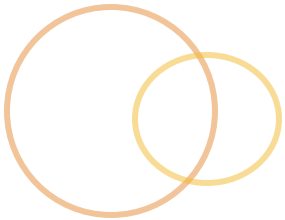
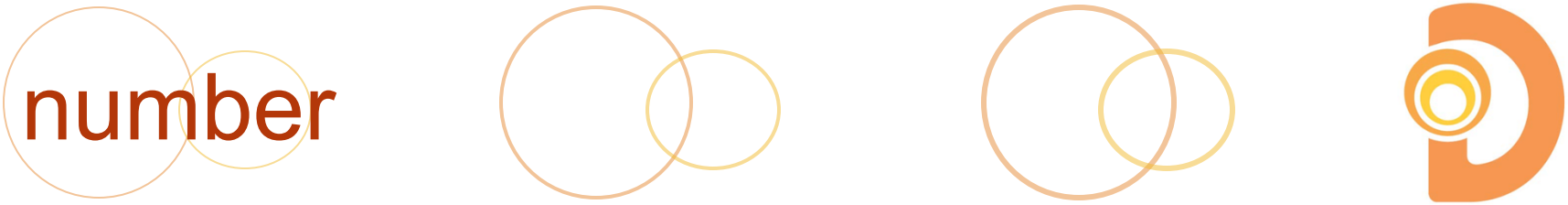
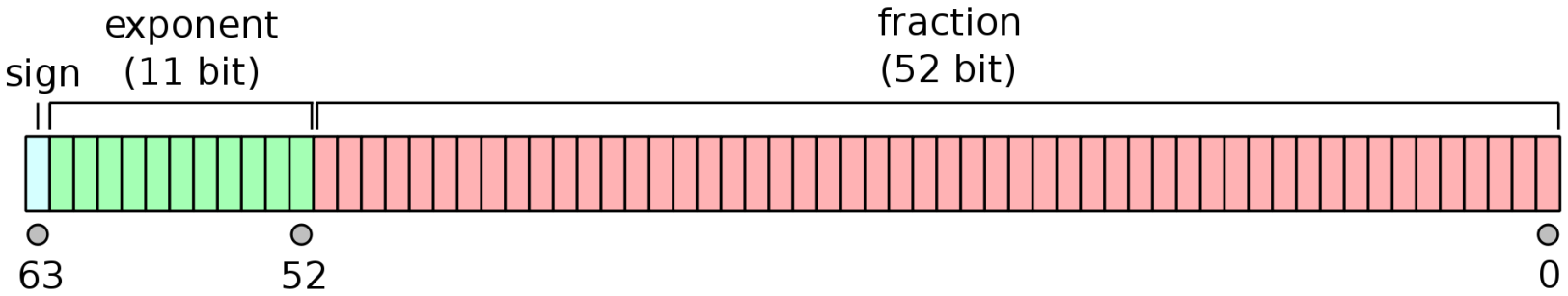
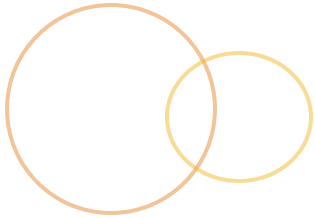
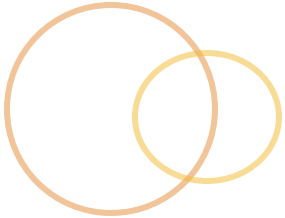
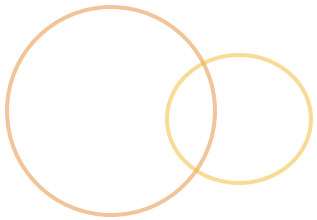
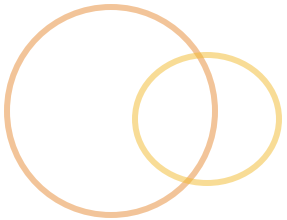
- Five *primitive* data types:
  - strings
  - numbers
  - booleans
  - null - lack of value
  - undefined – no value set, the default in JavaScript
  - *ES6: Additional primitive, Symbol*
- And then objects
  - Functions
    - “Invokable objects”
  - Built-in objects
    - String, Number, Boolean, Function, Array

Getting the type of a variable

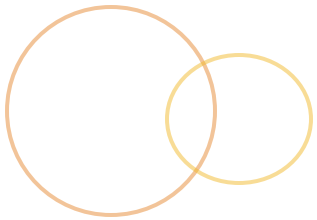
- The **typeof** keyword, followed by a variable, will return the primitive type of that variable

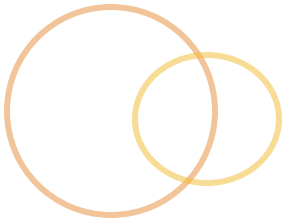
```
var π = 3.14;
typeof π; // 'number'
```

undefined & null

- Little difference between the two, in practice
- Variables declared without a value will start with `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



- 64bit floating point
- Numbers can be expressed as:
  - Decimal: **-9.81**
  - Scientific: **`2.998e8`**
  - Hexadecimal: **`0xFF;` `// 255`**
  - Octal: **`0777`**
  - `Binary:` **`0b10000000000000000000000000000000`**
- Special numbers:
  - `NaN`
    - `NaN == NaN; // false`
  - `Infinity and -Infinity`

number gotchas

- All numbers are stored internally as 64bit floating points



- Integers are accurate up to 15 digits

```
var y = 999999999999999; // 1000000000000000
```

- Decimals are accurate up to 17 digits

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

string





- Wrapped in single or double quotes
  - `var str = “My String”;`
  - `var str = 'My String';`
- Escape with backslash, \t and \n
- + operator for concatenation
  - `stringVar + ' ' + anotherStringVar + ' !';`
- Pro-tip:
  - Use + to coerce a string to a number
  - `(+“42”); // will equate to the integer 42`

object





- A list of zero or more pairs of keys and values, surrounded by curly braces
  - ```
    var obj = {bar: "baz"};
    obj.bar;      // "baz"
    ```
- Object keys
  - are non-sequential
  - need not be quoted, unless they contain characters that violate variable naming rules
- Values can be any type of data
- Can set/get via dot or [] array-like syntax

```
obj.foo = true;
obj['foo']      // true
```
```

arrays



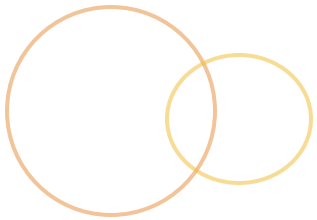


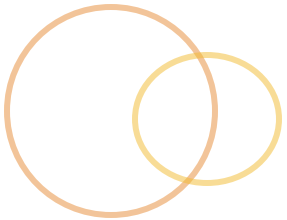
- Data stored *sequentially* with an index
- In JavaScript, the **array is an object** that behaves kinda like an array (*array-like*)

```
var emptyArray = [];
var myArray = [1,2,3,4];
myArray[1]; // 2
myArray[1] = 20;
```

- Strange behavior if you try to use string keys
  - var arr = [1,2,3];
    arr.length; // 3
    arr['bar'] = 10;
    arr.length; // 3

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
```

Exercise: Using Primitive Types

- Start the Node.js server

  ```
  node bin/server.js
  ```

- Open the following file:

  ```
  www/primitives/primitives.js
  ```

- Complete the exercise
- Run the tests by visiting in your browser:

  ```
  http://localhost:3000/primitives
  ```

Debugging in the console

- All modern browsers have built-in JS debuggers
- **console** object is a fairly standardized API
  - Typically on **window**
  - Methods like
    - log, warn, error
    - table(object)
    - group(name) *and* groupEnd()
    - assert(boolean, message)
- Can set breakpoints, monitor function scope, step in/out of calls, etc.

methods and properties

- Arrays objects have additional properties and methods
  - ```
    myArray.length; // 4
    myArray.push('John'); // adds value to end
    myArray.pop(); // John
    ```
- In fact, everything can act like an object…
  - ```
    var name= "John Smith";
    name.length; // 10
    ```
    "foo".toUpperCase(); // "FOO"
    5..toString(); // 5 ← wait, what?

Everything* is an object

- *most things, primitives are just coerced
- Primitive literals all have Object counterparts
  - except null and undefined

```
5 === Number("5");
"Hello" === String("Hello");
true === Boolean(1);
```

- Temporarily coerced to object when used as object
- So… we can access properties and invoke methods of objects, including primitives
  - `str.length;`
  - `str.toUpperCase();`
  - `“Hello”.length;`

Built-in constructors vs literals

- Literals constructed by their object counterpart

```
new String("Hi"); // {0: "H", 1: "I"
String("Hi"); // "Hi"
new Number(5); // 5
new Array(1,2,3); // [1,2,3]
new Boolean(1); // true
new Object(); // {}
```

- Cons
  - Wrapped by object indefinitely
  - typeof can return"object"
  - Behavior changes based on new usage
  - Array(x) does weird things

Type Coercion

- 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"
```

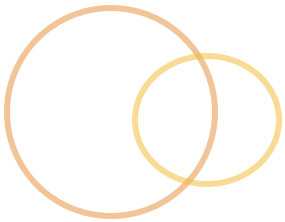
Implicit Coercion

- It's not obvious how it will coerce…
  - `8 * null -> 0`
    `“5” – 1 -> 4`
    `“5” + 1 -> 51`
- Much confusion ensues
  - `[] + [] -> empty string`
  - `[] + {} -> [object Object]`
  - `{} + [] -> 0`
  - `{} + {} -> NaN`

Recap: basic data types

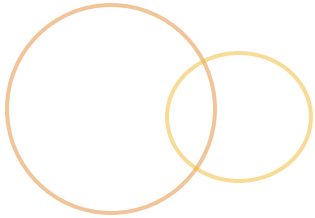
- There are 5 primitive types (string, number, boolean, null, undefined) and then Objects
  - Functions are a callable Object
  - Objects are property names referencing data
  - Arrays are for sequential data
- Declare variables with "var"
- Types are coerced
  - Including when a primitive is used like an object
- *Almost Everything* is an object, except the primitives
  - despite them having object counterparts

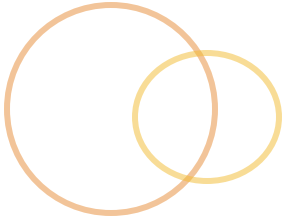
Module: Operators



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

Unary



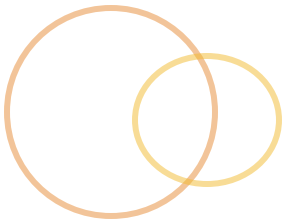


```
delete obj.x    // undefined
void 5 + 5      // undefined
typeof 5        // 'number'
+'5'            // 5
-x              // -5
~9              // -10
!true           // false
++x             // 6
x++             // 5
--x             // 4
x--             // 5
```

Shortcuts

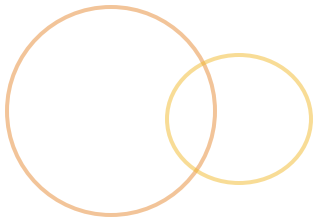
- For your bag of tricks:
  - `(+x);`
    - Convert string to a number
  - `!!myVar;`
    - Double bang can convert any value to a boolean

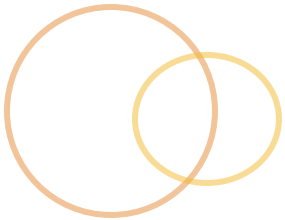
typeof operator



- Takes one operand
  - `typeof variableName;`
- Returns a string
- Type gotchas
  - object = object
    array = object
    number = number
    boolean = boolean
    undefined = undefined
    string = string
    function = function
    but… null = object
  - http://jsfiddle.net/mrmorris/DjBtL/

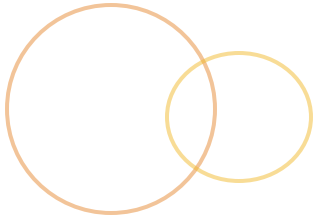
Arithmetic

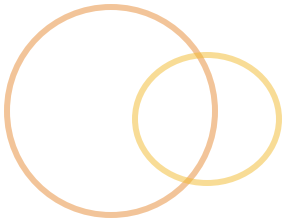




```
5 + 5          // 10
5 - 3          // 2
5 * 2          // 10
10 / 2         // 5
10 % 3         // 1
```

Bitwise



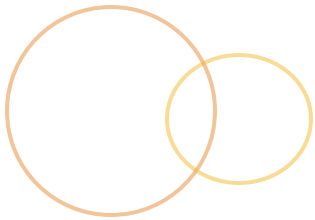


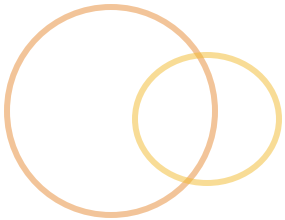
```
5 & 4    // 1
1 | 4    // 5
4 ^ 6    // 2
9 << 2   // 36
-9 >> 2  // -3
9 >>> 2  // 2
```

Assignment

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

Relational



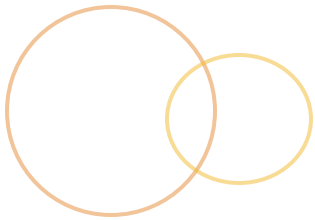


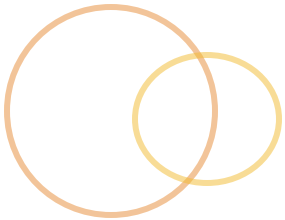
```
'foo' in {foo: true}  // true
[] instanceof Array   // true
5 < 4                 // false
5 > 4                 // true
4 <= 4                // true
5 >= 10               // false
```

Equality* (strict vs loose)

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

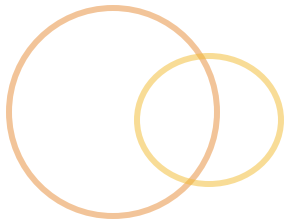
Logical





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

Logical short circuits



- **a && b** returns either a or b

```
if (a) {
    return b;
} else {
    return a;
}
```

- **a || b** returns either a otherwise b

```
if (a) {
    return a;
} else {
    return b;
}
```

Exercise: Boolean Operators

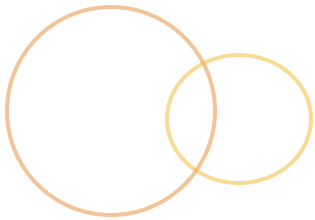
- Experiment with &&:

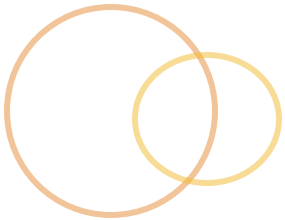
```
false && console.log("Yep");
true && console.log("Yep");
```

- Experiment with ||:

```
false || console.log("Yep");
true || console.log("Yep");
```

Ternary





```
// condition ? then : else;
true ? 'foo' : 'bar' // 'foo'
```

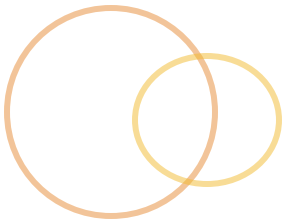
Falsy / Truthy

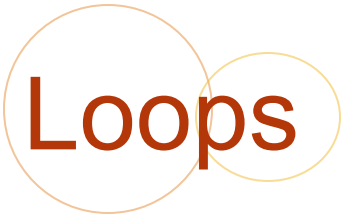
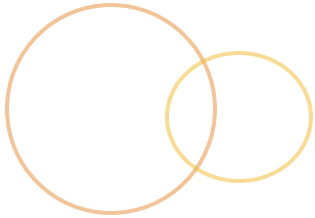
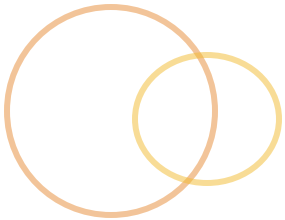
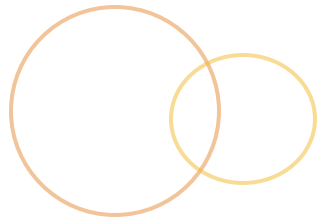
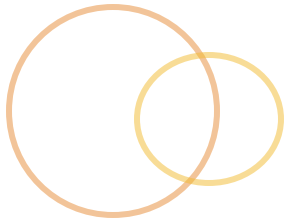
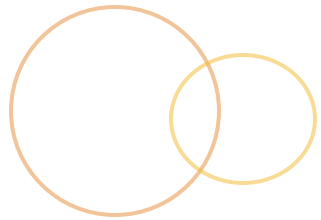
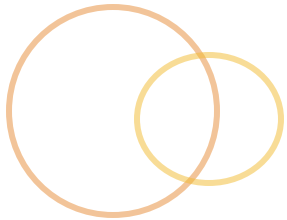
- These are **falsy**
  - `false`
  - `null`
  - `undefined`
  - `""`
  - `0`
  - `NaN`
- Everything else is **truthy**, including…
  - {}
  - []
  - "0"
  - "false"

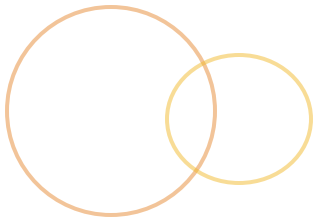
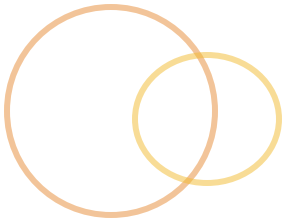
Module: Control Structures

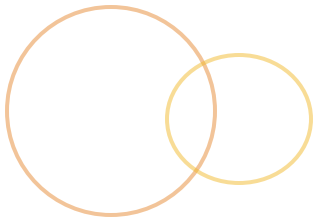
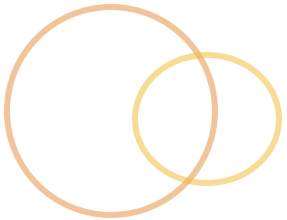
- Conditionals
- Loops
- Flow Control
- Labels

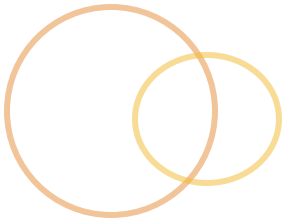
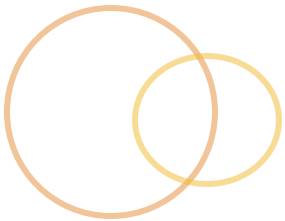
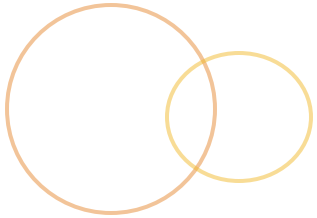
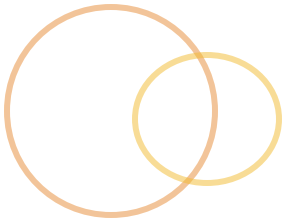
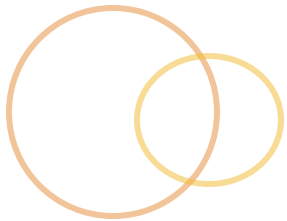
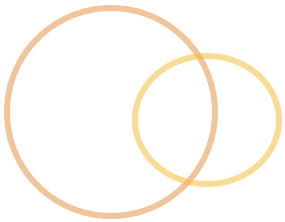
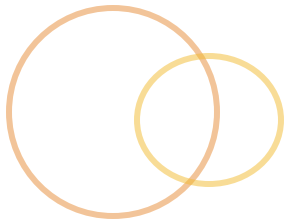
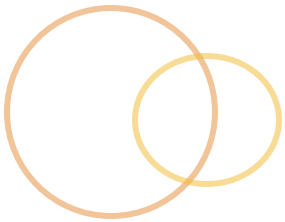
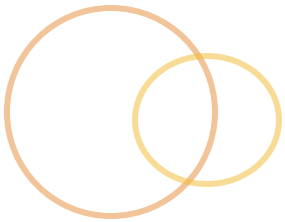
Conditional statements

- `if (expression) {…}`
- ```
  if (expression) {
      …
  } else {
      …
  }
  ```
- `if {} else if {} else {}`
  ```

Switch statements



```
switch (expression) {
    case val1:
        // statements
        break;

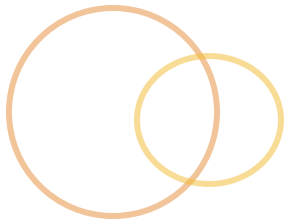
    case default:
        // statements
        break;
}
```

Loops






- `for`
- `while`
- `do…while`
- `for…in`
- ~~`for each…in`~~
- ~~`with`~~

For loop





```
for (var i=0; i<10; i++) {
    // executes 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
  - Use `obj.hasOwnProperty(propertyName)`

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

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

For each in...



- ~~For each...in~~
  - deprecated
  - Could iterate over property ***values***
  - `for each (var value in object) {}`
- ES6: has for…of
  - for (let value of iterable) {
    value += 1;
    console.log(value);
    }

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);
    }
}
```

Control Structures Recap

- Conditionals like if and if-else
- Switch statements
- Iterate (loop) with while and for
- Enumerate over an object with for..in
- Examples
  - http://jsfiddle.net/mrmorris/GN7qL/

Exercise: Control Flow

- ~~Start the Node.js server~~

  ~~`node bin/server.js`~~

- Open the following file:

  `www/control/control.js`

- Complete the exercise
- Run the tests by visiting in your browser:

  `http://localhost:3000/control/`

Exercise – Control flow

- Get to know control flow and iteration statements
- We'll use some basic browser functions
  - `alert("A message!");`
  - `var response = prompt("Ask for a value!");`
  - `confirm("Ask user to say 'ok'");`
- Fork me
  - http://jsfiddle.net/mrmorris/cxz2hta1/

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: "The best part of JS"

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

Defining a function



- Three ways
  - Function **declaration**
  - Function **expression**
  - with the `Function()` **constructor**
    - `new Function (arg1, arg2, … argn, functionBody)`
- http://jsfiddle.net/mrmorris/N8vcg/

Function Declaration



```
function adder(a, b) {
     return a + b;
}
adder(1, 2); // 3
```

- The function name is mandatory
- Function statements (declarations) are ***hoisted*** to the top of the scope, therefore are not great for dynamically created functions

Function Expressions



- Define a function and assigns it to a variable
  - ```
    var adder = function(a, b) {
     return a + b;
    }
    ```
- Name is optional in a function expression
  - ```
    // stores ref to anonymous function
    var funcRef = function() {};
    ```
  - ```
    // stores ref to named function
    // *name is scoped to inner function
    var funcRef = function funcName() {};
    ```
    ```

Function Invokation

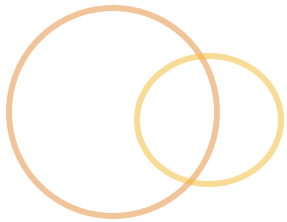


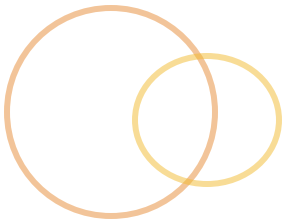
- Invoke with `()`
  - `myFunctionName(argument1, argument2);`
- Any number of arguments can be passed in, regardless of defined parameters
- Missing arguments are set as `undefined`

Function argument defaults

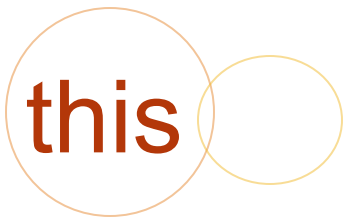
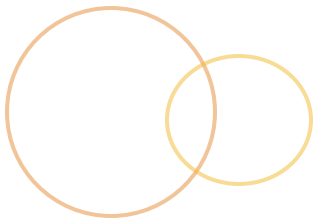
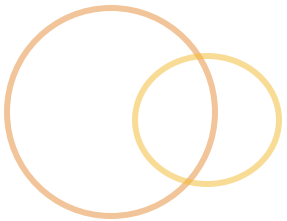
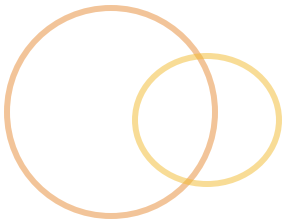
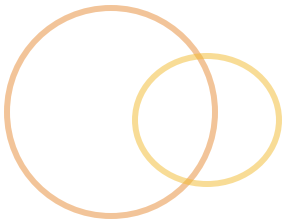
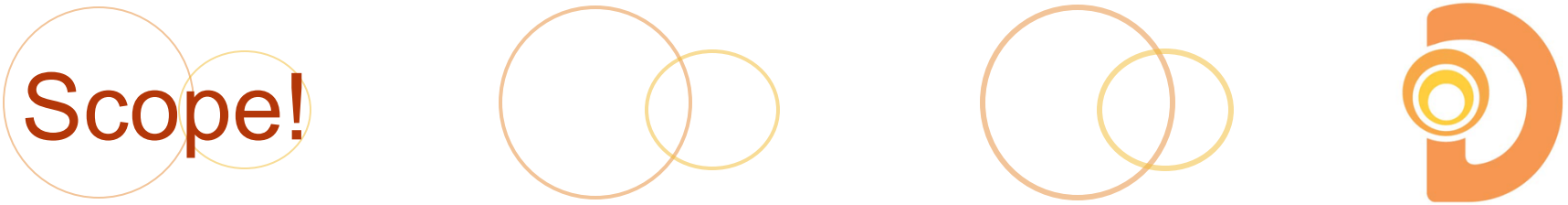
- ES5 and below
  - function adder(a, b) {
    a = a || 0;
    b = b || 0;
    return a + b;
    }
- ES6 supports default values
  - function adder(a = 0, b = 0) {
    return a + b;
    }

Return statements



- `return <expression>;`
- Will return the result of the expression as a result of the function invokation, to the caller of the function
- Constructor functions will return `this`
- Careful with your line breaks…
  - ```
    return
      x;
    // Becomes
    return;
      x;
    ```
    ```

Function pseudo-parameters

- Every function has access to special parameters upon invokation
  - `arguments`
  - `this`

arguments



- Includes all arguments passed to the function
  - ```
    function test(a) {
      arguments[0] === a; // true
    }
    ```
- Array-like, but not an actual array
  - Only has `length` property
- “live”; can affect actual arguments
  - Should be treated as read-only
    ```

Arguments as an array

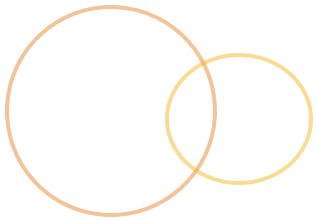
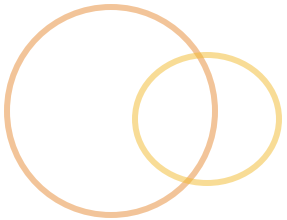
- Arguments is array-like so it doesn't have all the array functionality you may want
- To treat like an array, convert it to one…
  - ```
    var arr = Array.prototype.slice.apply(
     arguments
    );
    ```
- http://jsfiddle.net/mrmorris/2kpyB/
    ```

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>;
```

- Careful with your line breaks…

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

Global functions



- `alert(msg);`
- `confirm(msg)`
- `prompt(msg, msg);`
- `isFinite()`
- ~~`isNaN()`~~
- `parseInt()`
- `parseFloat()`
- `encodeURI(), decodeURI()`
- `setInterval, clearInterval`
- `setTimeout, clearTimeout`
- `eval(); // dangerous`

Exercise: Function Arguments

- ~~Start the Node.js server~~

  ~~`node bin/server.js`~~

- Open the following file:

  `www/parse/parse.js`

- Complete the exercise
- Run the tests by visiting in your browser:

  `http://localhost:3000/parse/`

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

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
  - Scope is defined by author
  - Where a ***var*** is declared
  - Can be determined by reading (no need to execute)
- 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.
  - ES6 supports block-scope with ***let*** keyword

Example: Identify the scope

```
var a = 5;

function foo(b) {
        var c = 10;
        d = 15;

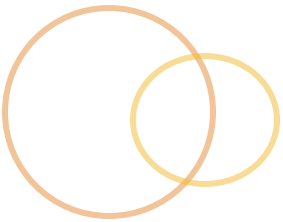
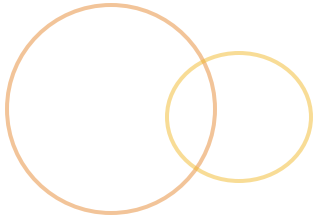
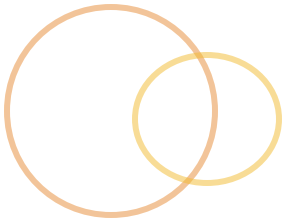
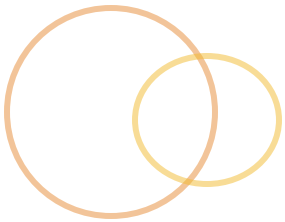
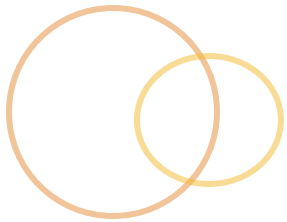
        if (d < b) {
                var e = 'Hoisty McHoisted';
        }

        function bar(f) {
                var c = 2;
                a = 12;
        }
}
```

Scope matters

- Global vs Local
  - **window** is global scope in the browser
- 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…

- http://jsfiddle.net/mrmorris/YJL9u/

No block scope…

- I lied…
  - eval() can cheat scoping, inserting itself into the scope
    - But… in *strict mode* eval() creates its own scope
  - "with" simulates some block scope, but not all – deprecated
  - An exception's "catch" block has its own scope

Exercise: Hoisting (part 1 of 2)

- What will the output be?

```
function foo() {
  x = 42;
  var x;

  console.log(x);
}
```

Answer: Hoisting (part 1 of 2)

- Becomes

```
function foo() {
  var x;
  x = 42;

  console.log(x);
}
```

Exercise: Hoisting (part 2 of 2)

- What will the output be?

```
function foo() {
  console.log(x);
  var x = 42;
}
```

Answer: Hoisting (part 2 of 2)

- Becomes:

```
function foo() {
  var x;
  console.log(x);
  x = 42;
}
```

Hoisting





- *Hoisting* refers to when a variable declaration is lifted and moved to the top of its scope
  - … only the declaration, not the assignment
- function *statements* are hoisted, too, so you *can* use them before actual declaration
- JavaScript essentially breaks a variable declaration into two statements
  - ```
    var myVar=0, myOtherVar;
    // is interpreted as
    var myVar=undefined, myOtherVar=undefined;
    myVar=0;
    ```
- http://jsfiddle.net/mrmorris/dcvn6735/
    ```

Scope continued

- Example of basic 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
      }
    }
    ```
  - Three scopes exists here
- Each inner scope has access to the outer, *but* the outer scopes cannot access inners
- ReferenceError indicates unfound in scope chain
- http://jsfiddle.net/mrmorris/YJL9u/
    ```

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 so far…

- 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()
- Include implicit ;
- Always open and close blocks with { }
- Indent and empty lines ensure readability

End of day 1?

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?

Module: Objects, pt 2



- 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

Creating an object literal

- Create an object literal with {}:

```
var myObjLiteral = {
    name: “Mr Object”,
    age: 99,
    toString = function() {
        return this.name;
    }
};
```

- http://jsfiddle.net/mrmorris/4dsLonat/

Object properties



- Four ways to access and set
  - Dot Syntax
    - myObj.key;
  - Square bracket Syntax
    - myObj[“key”];
  - `Object.defineProperty(myObj, “key”, desc)`
    - Takes a descriptor defining property properties
  - `Object.defineProperties(myObj, “key”, desc)`
- Can delete a property with `delete`
  - `delete myObj.key;`

Properties descriptors

- Object properties have (usually) hidden descriptors that affect its behavior
  - `enumerable`
  - `configurable`
  - Optional…
    - `value`
    - `writable`
    - `get`
    - `Set`
- ```
  Object.defineProperty(obj, “key”, {
      .. descriptors ..
  }
  ```
  ```

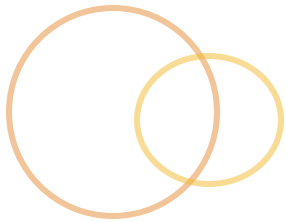
Object reflection

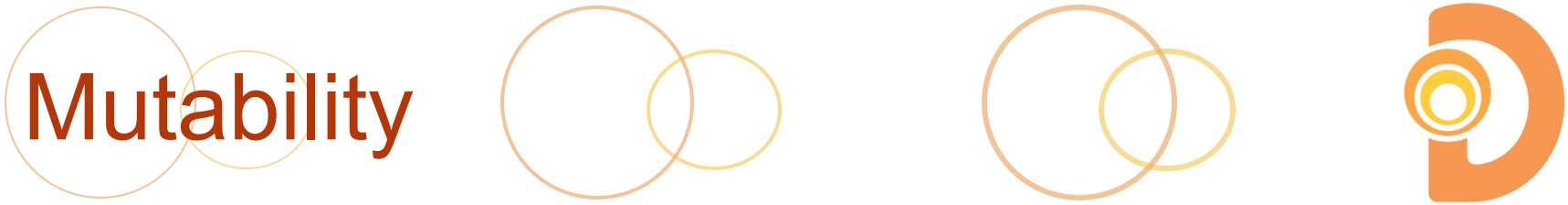
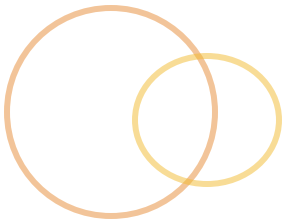
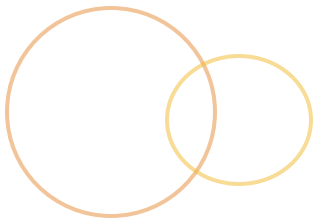
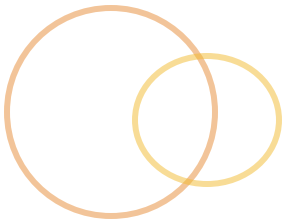
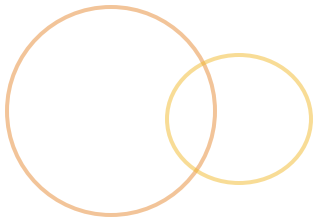
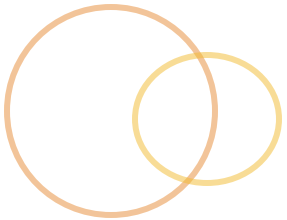
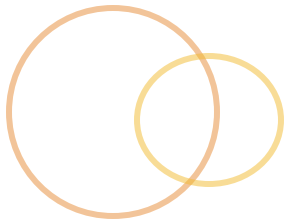
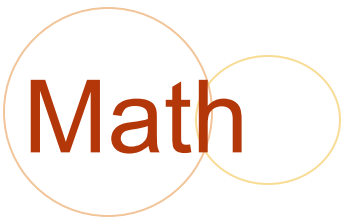
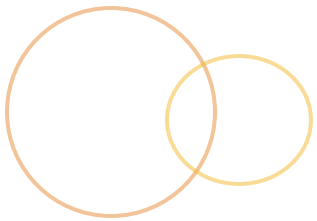
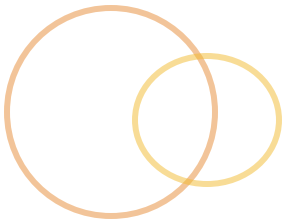
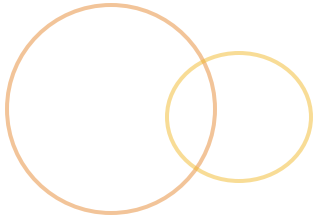
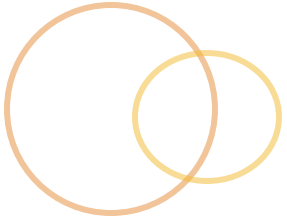
- `typeof`
- `in` operator
  - `“propertyName” in object`
  - Goes all the way up the chain, not just “own”
- `hasOwnProperty`
  - `myObj.hasOwnProperty(“propertyName”)`
- Keep in mind that objects "inherit" properties. Use **hasOwnProperty** to see if an object actually has it's own copy of a property.

Object

- All built-in objects derive from the Object object
- Properties and methods on Object are **inherited** to all other Objects
  - Prototypal inheritance
- 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

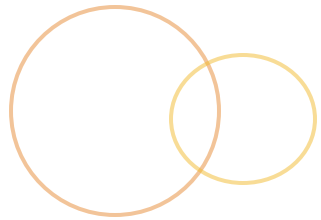
Object enumerating

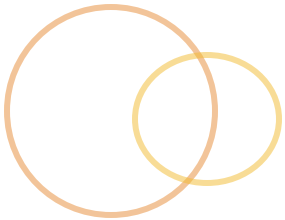


- `for…in`
  - ```
    for (var propName in objectName) {
     objectName[propName];
    }
    ```
  - Enumerates over enumerable properties
  - And all *inherited* properties
  - Arbitrary order (not for arrays)
    ```

Object enumeration, continued

- `Object.keys(obj)`
  - Returns array of all “own”, enumerable properties
- `Object.getOwnPropertyNames(obj)`
  - Returns array of all own property names, including non-enumerable

Call-by-sharing

- JavaScript is **call-by-sharing** (similar to call-by-reference)
- Objects in a call-by-sharing world
  - Passing an object as an argument to a function
    - Objects are mutable
    - Re-assigning the variable does not affect the object
  - Comparing objects with == vs ===
    - === expects the objects to be the actual same object in memory
- http://jsfiddle.net/mrmorris/ZLW22/

Mutability



- All primitives in JavaScript are immutable
  - Using an assignment operator just creates a new instance of the primitive
  - Pass-by-value
  - Unless you used an object constructor for a primitive…
- Objects are mutable (and pass-by-reference)
  - Their values (properties) can change

Exercise: Copying objects

- ~~Start the Node.js server~~

  ~~`node bin/server.js`~~

- Open the following file:

  `www/copy/copy.js`

- Complete the exercise
- Run the tests by visiting in your browser:

  `http://localhost:3000/copy/`

- `Hint: for (var prop in obj) { /* */ }`
- `Hint: obj.hasOwnProperty(prop)`

Exercise: Objects



- Create a “copy” function
- Fork me:
  - http://jsfiddle.net/mrmorris/mLccst8c/

Built-in Objects

- String
- Number
- Boolean
- Function
- Array
- Date
- Math
- RegExp
- Error
- http://jsfiddle.net/mrmorris/rrb67ev0/

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
```

Number Properties



- Properties
  - `MAX_VALUE`
  - `NaN`
  - Etc…
- Generic methods
  - Number.isInteger()
  - Number.isFinite()
  - Number.parseFloat()
  - Number.parseInt()
- Instance methods
  - `num.toString()`
  - `num.toFixed()`
  - `num.toExponential()`

Math






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

Arrays





- Arrays are objects that behave like arrays
  - Indexes of an array are converted to strings and set as object properties
- Use arrays when order of the data should be sequential
- Creating an array
  - ```
    // literal
    var myArray = [1,2,3];

    // constructor
    var myArray = new Array(1,2,3);
    ```
- http://jsfiddle.net/mrmorris/at6djrey/
    ```

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]
```

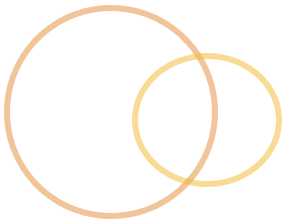
Array accessor methods

```
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); // new array filtered
arr.map(fn); // new array transformed
arr.reduce(fn); // result from an array
arr.reduceRight(fn);
```

Array enumeration

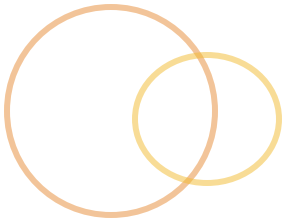


- Use “`for`” not “`for…in`”, which doesn’t keep array keys in order
  - `for (var i=0; i < myArray.length; i++) {`

    `}`
- `.forEach(callback[, thisArg])`
  - New in es5 (ie9+)
  - No way to stop or break a `forEach` loop
  - `myArray.forEach(function(val, index, arr) {`

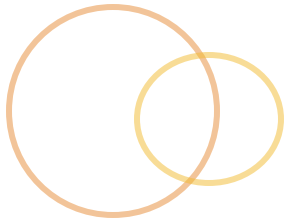
    `});`

Array tests



- `arr.every(callback[, thisArg])`
  - checks If every element in an array passes a callback function
  - ```
    myArray.every(function(val, index, arr) {
     return (val>0); // evaluates to boolean
    });
    ```
- `arr.some(callback[, thisArg])`
  - verify if at least one passes the test
    ```

Array filter, map

- `.filter()`
  - Iterate over your array of items passing them to a function. Returning `true` from the function indicates the item should be retained.
    - ```
      myArray.filter(function(item) {
            return item!=2;
      }); // removes items that don't equal 2
      ```
- `.map()`
  - Iterates over array, invoking a function on each value. The return value is the modified value of the item.
- http://jsfiddle.net/mrmorris/pbwy3hy5/
      ```

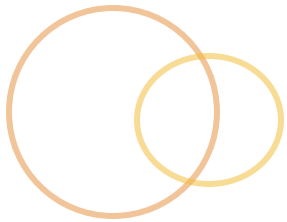
Array reduce()



- `.reduce()`
  - Boils down a list of values into a single value.

```
[0,1,2,3,4].reduce(function(acc, elm) {
     // 1. acc is the accumulator
     // 2. elm is the current element
     // 3. You must return a new accumulator
     return acc + elm;
}, 0);
// initial acc value can be passed in
```

Exercise: Arrays



- ~~Start the Node.js server~~

  ~~`node bin/server.js`~~

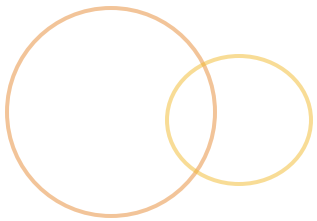
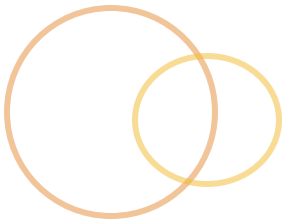
- Open the following file:

  `www/array/array.js`

- Complete the exercise
- Run the tests by visiting in your browser:

  `http://localhost:3000/array/`

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

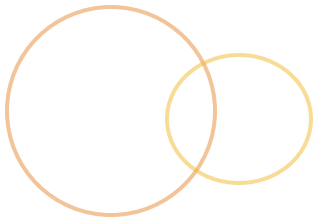
- Generics

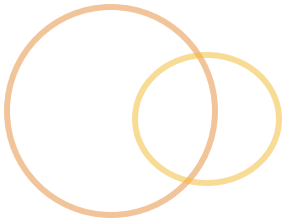
```
Date.now()
Date.parse('2015-01-01')
Date.UTC(2015, 0, 1)
```

- Instance method examples

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

RegExp





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

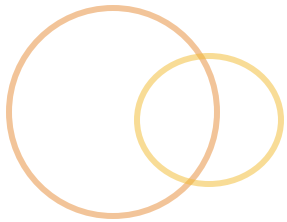
```
var re = new RegExp("\w+", "g");
var re = /\w+/g;
```

- 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

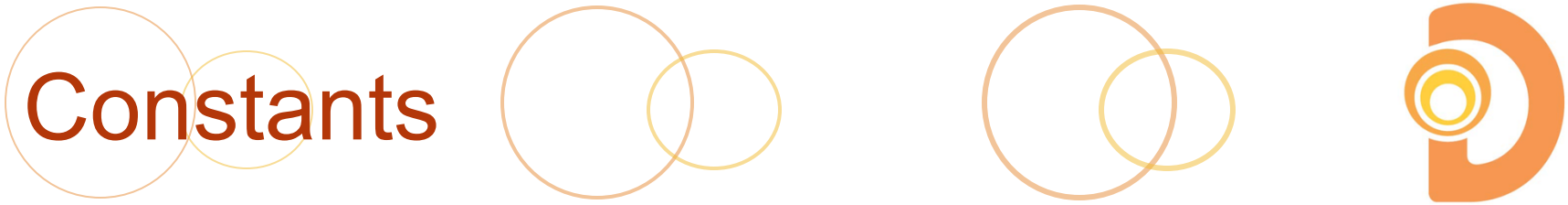


- Instance methods
  - re.**exec**(*str*)
  - re.**test**(*str*)
- String methods that accept RegExp params
  - str.**match**(*regexp*); // array of matches
  - str.**replace**(*regexp, replacement*); // string with replacement
  - str.**search**(*regexp*); // returns 1 at first match
  - str.**split**(*regexp, limit*); // returns array

Regular Expressions matchers

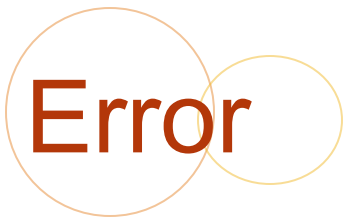
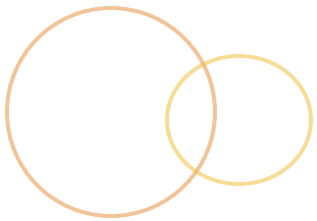
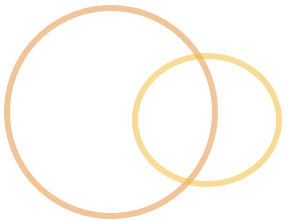
- Escape with / backslash
- Use [] for sets, [01234] or [0-4] for a range
- Special character groups, ex: \d (digits) and \w (alphanumeric)
- + match at least one
- * match 0 or more
- ^ invert
- ? Optional - /neighbou?r/
- {4} time occurrence
- {2,4} – at least twice, at most 4 times

Constants



- ES6, Not frequently used
- Immutable while script thread is running
- Same rules as apply to variables, but keyword ‘`const`’ is used instead of ‘`var`’
- They *are* scoped

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!');
```

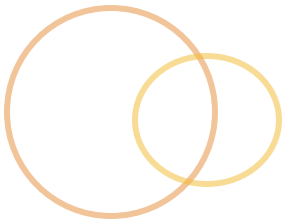
- Implementation varies across vendors
- Instance properties

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

Error Handling

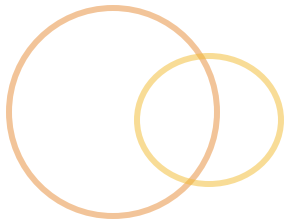
- JavaScript is very lenient when it comes to handling errors
- Internal errors are raised via the **throw** keyword, and are then considered "exceptions"
- Exceptions are handled via a **try/catch/finally** construct, where the thrown exception is passed to the **catch** block
  - Nesting allowed
  - 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

Built-in Errors



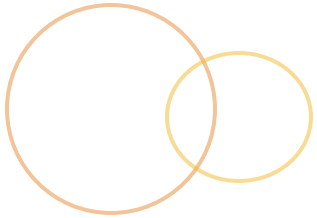
- Error (Top level object)
- SyntaxError
- ReferenceError
- TypeError
- RangeError
- URIError
- EvalError

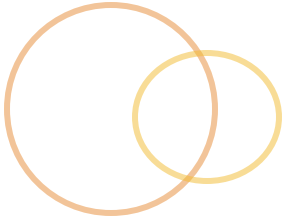
The global object



- The scope in which all variables live by default
- The context of the application
- “window” when running in a browser

ES5…



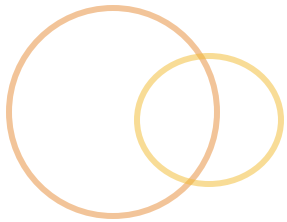


- Ignore trailing commas
- No reserved word restrictions on prop names
- Built in getter/setter property settings
- Infinity, NaN and undefined are now constants
- JSON is now native
- Now has a “strict mode”
- Many new methods!
  - Function.prototype.bind()
  - String.prototype.trim()
  - Array.prototype.every(), filter, map, reduce
  - Object.keys()
  - Object.create()
  - Array.isArray()

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/mrmorris/d2f6hohb/

Object immutability



- Object.freeze(obj)
  - Freezes an object: other code can't delete or change any properties
  - http://jsfiddle.net/mrmorris/auuzgaxr/
- Object.preventExtensions(obj)
  - Prevents any extensions of an object
  - http://jsfiddle.net/mrmorris/0mn20b5L/
- Object.seal(obj)
  - Prevents other code from deleting properties of an object
  - http://jsfiddle.net/mrmorris/jmqguegw/
- Object.defineProperty(obj, propName, definition)
  - Defines (or updates) a property on an object
  - http://jsfiddle.net/mrmorris/g6t0hywp/

Exercise – Core objects

- Reverse an array
  - http://jsfiddle.net/mrmorris/zqfwy1xp/
- Strings and Dates
  - http://jsfiddle.net/mrmorris/vk2tg472/
- Data grids in the console
  - http://jsfiddle.net/mrmorris/0kptbv7p/
- Don’t forget devdocs.io!

Exercise: String Manipulation

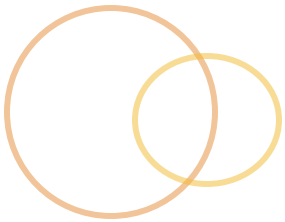
- ~~Start the Node.js server~~
  ~~`node bin/server.js`~~
- Open the following file:
  `www/string/string.js`
- Complete the exercise
- Run the tests by visiting in your browser:
  `http://localhost:3000/string/`

- Hint: Use `http://devdocs.io` or `https://developer.mozilla.org /` for docs

Module: Function patterns

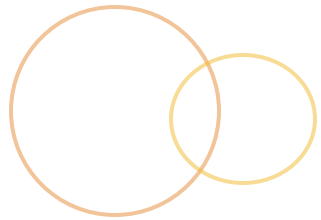
- Closures
- Immediately invoked functions
- Anonymous Functions
- …Lots more we won't cover (chaining, currying, lazy load)

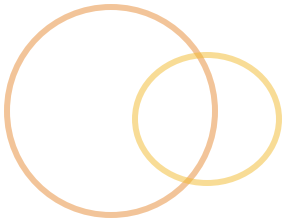
Anonymous functions



- A function expression
  - `var anon = function anon() {}`
- Pros
  - Functions can be passed as arguments
  - Defined inline
  - Supports dynamic function definition
  - Can be named, which is scoped to function
- But…
  - difficult to test in isolation
  - Discourages code re-use
  - Hard to debug (unless you *name* it)

IIFE



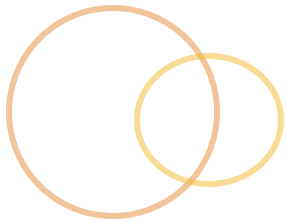


- Immediately Invoked Function Expressions
- Define the function and immediately invoke it
  - var tasksController = function() {
    // body
    }(); // there it is!
- Anonymous? Sure…
  - (function(){
    …
    }());
  - (function(){
    …
    })(); // also works
- http://jsfiddle.net/mrmorris/u5srzzgk/

Closures

- One of the most important features of JavaScript
- And often one of the most misunderstood & feared features
- But… they are *all around you* in JavaScript
- They happen when you write code that relies on lexical scope

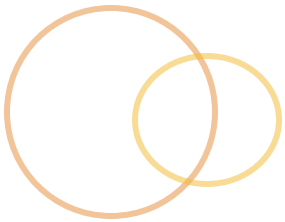
Closures Example



```
function closeOverMe() {
     var a=1;
     return function() {
          console.log(a);
     };
};
var witness = closeOverMe();
witness(); // 1
```

- More examples
  - http://jsfiddle.net/mrmorris/974U2/

Closures for Privacy

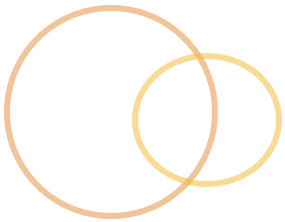


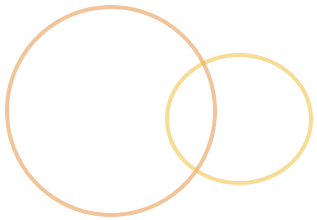
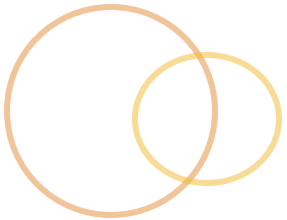
```
var controller = function() {
        var privateVar = 42;

        var getter = function() {
                return privateVar;
        }


        return {
                getPrivateVar: getter
        }
}
var x = Controller();
```

Closures, continued

- “When the context of an inner function includes the scope of the outer function and the inner function enjoys the context even after the parent function has returned”
- When a function is able to remember and access its lexical scope, even when executing outside its lexical scope.
- When an inner function *closes over* the variables of an outer function
- It retains state and scope after it completes execution
- And… the inner function is used outside the outer

Function callbacks

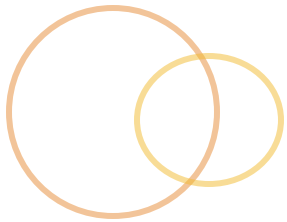


- When a function is provided as an argument as something to be invoked inline, or under specific circumstances (like an event)
- ```
  function runCallback(callback) {
      // does things
      return callback();
  }
  ```
  ```

Timers





- Establish delay for function invokation
  - `setTimeout(func, delayInMs[, arg1, argn])`
    - `var timer = setTimeout(func, 500);`
  - Use `clearTimeout(timer)` to cancel
- Establish an interval for periodic invokation
  - `setInterval(func, ms)`
  - `clearInterval(timer)`
- Context will always be global for the callbacks
- http://jsfiddle.net/mrmorris/s5g2moc6/

Callbacks and closures

- Careful with function expressions in loops
  - Can have scope issues
    - ```
      for (var i=0; i<3; i++) {
            setTimeout(function(){
                  console.log(i);
            }, 1000*i);
      } // what will this output?
      ```
  - Instead, create an additional scope to maintain state for the inner function (expression)
- Closures save the day
  - http://jsfiddle.net/mrmorris/e8n62r3w/
      ```

Function patterns recap

- Mind your **scope**!
  - In callbacks, in particular
- **Closures** create a persistent and private scope
- Functions are often passed around as **callbacks** (due to the nature of the event loop…)
- Function expressions can be **immediately invoked** (and you'll see this a lot)

Exercise: Closures



- ~~Start the Node.js server~~

  ~~`node bin/server.js`~~

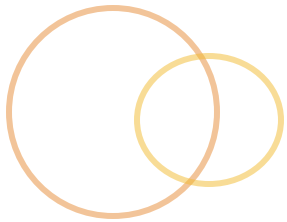
- Open the following file:

  `www/closure/closure.js`

- Complete the exercise
- Run the tests by visiting in your browser:

  `http://localhost:3000/closure/`

Exercise: Closures

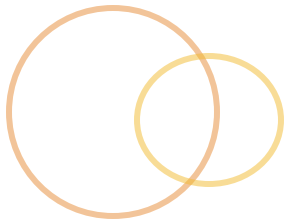


- Month names function
  - http://jsfiddle.net/mrmorris/y37qch2g/

Module: Scope & Context

- We already discussed *Scope*
  - Determines visibility of variables
  - Lexical scope (write-time)
- There is also *Context*
  - Context refers to the location a function/method was invoked from
  - Sort of dynamic scope; defined at run-time
  - Context is referenced by “this”

“this” is the context



- `this` keyword is a reference to the “object of invokation”
  - Bound at invokation
  - Depends on the call-site of the function
- It…
  - allows a method to know what object it is concerned with
  - allows a single function object instance to service many functions/usages
  - is key to inheritance
  - gives methods access to their objects

“this” is determined by call-site

- this is *bound* at call-time to an object
- 1) Default binding
  - Global
- 2) Implicit binding
  - Object method
    - Warning: Inside an inner function of an object method it refers to the global object
- 3) Explicit binding
  - Set with .call() or .apply()
- 4) Hard binding
  - Defined by .bind()
- 5) “new” binding
  - When *constructing* a new object
- http://jsfiddle.net/mrmorris/RUNS5/

More explicit binding

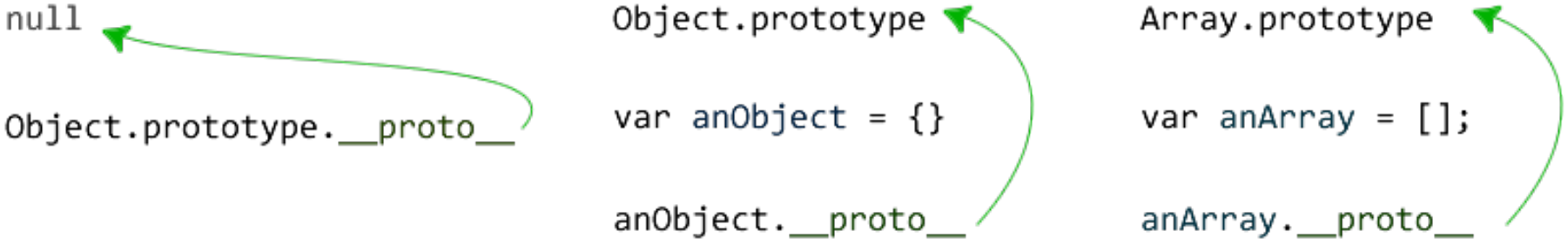
- Context can be changed, which affects the value of `this`, via Function's `call`, `apply` and `bind` methods
  - ```
    obj.foo(); // obj context
    obj.foo.call(window); // window context
    ```
- "`bind`" doesn't execute, it returns a copy of the function with the context re-defined. The resulting function is a "bound function"
  - ```
    var getX = module.getX;
    boundGetX = getX.bind(module);
    ```
- http://jsfiddle.net/mrmorris/or7y5orn/

Inheritance in JavaScript

- Or.. How I learned to love the Prototype
- JavaScript is class-free, prototypal
- There is a way to simulate classical inheritance, but prototypal approach is more suited to JavaScript
- Prototype modal
  - Tends to be smaller
  - Less redundant
  - Can simulate classical inheritance as needed
  - More powerful

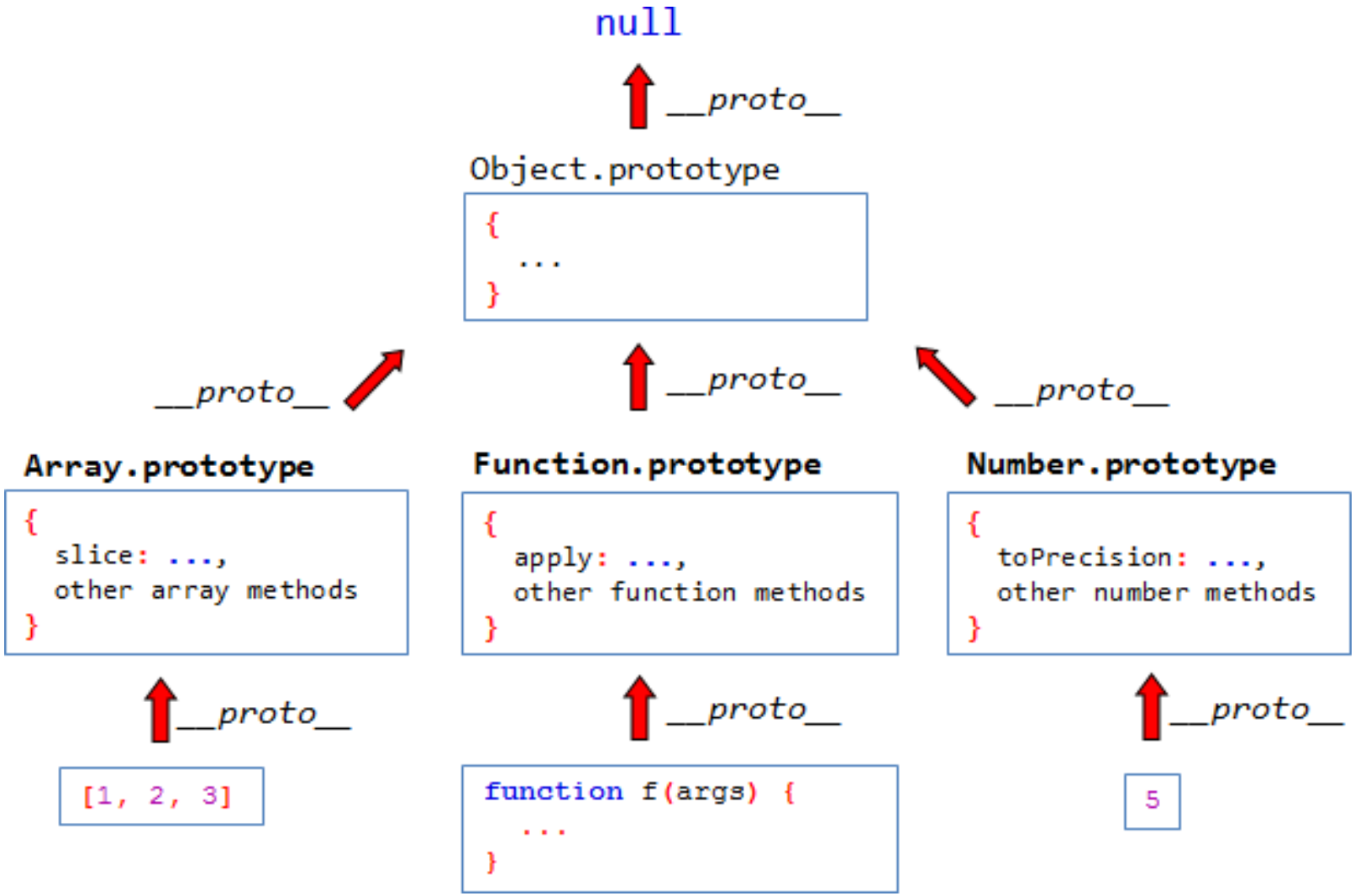
Prototypes (intro)

- **Objects** have an internal link to another object called its ***prototype***
- That *prototype* has a *prototype*, and so on, up the ***prototype chain***



- The chain goes up to **Object.prototype** and then **null** (the end of the chain)
- Objects ***delegate*** behavior to other objects through this prototype linkage

This is how objects inherit their stuff



Inheritance Patterns in JavaScript

- Two Schools
  - Classical
  - Prototypal
- Several patterns to familiarize with
  - Native constructor…
  - Pseudo-classical
  - *Prototypal
  - Parasitic

Let's return to objects…

- There are three common ways to create new objects in JavaScript
  - Object Literal
    - `var newObj = {};`
  - Object.create(prototypeObj) - *ES5*
    - `var newObj = Object.create(Object.prototype);`
  - Constructor Function
    - `var newObj = new Object();`

Native Constructor Pattern

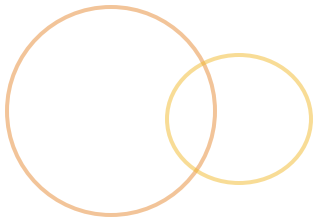
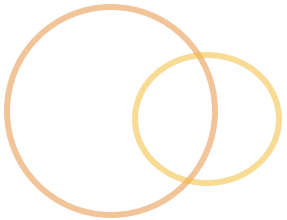
- Tends to simulate classical inheritance
- Relies on "`new`" keyword and "`this`"

```
function Car(color) {
  this.color = color;
  this.toString = function() {
    return this.color + " car";
  };
};
var toyota = new Car("red");
```

- It…
  - Creates a new object
  - Invokes the function with that object as the context
  - Sets the prototype of the new object to the Constructor's prototype
  - Returns the new object

Native Constructor continued

- Can make the Car instances more flexible by utilizing its prototype property
  - ```
    function Car(color) {
     this.color = color;
    }

    Car.prototype.toString = function() {
     return this.color + " car";
    }

    var toyota = new Car("red");
    // toyota.__proto__ -> Car.prototype
    ```
    ```

Augmenting

- The **prototype** behaves like a live template for objects that link to it in the chain
- (*In the Constructor Pattern*) you can continue to augment instances through its Constructor's prototype
  - `Car.prototype.drive = function(){}`

```
// now toyota will immediately have this
// available

toyota.drive();
```

Own properties

- An object can have its “own” properties, referenced within the object itself
- It can also have properties it delegates up the prototype chain
- Continuing our Car example…
  - toyota => {color: “red”}
  - toyota.red; // is it’s own property (thanks to the constructor)
  - toyota.drive(); // is delegated up the chain
  - Car.prototype.tires = 4; // now augment
  - toyota.tires; // works through delegation up the chain

Prototype





- Every function in JavaScript has a `prototype` property, which is by default an empty object.
- Every object in JavaScript has a internal reference to it's prototype (`__proto__`)
  - Better said as, "A reference to the object's prototype that it inherits from"
- Objects also have a `constructor` property
  - What constructor function made me?

prototype vs. __proto__

- **.prototype** is a property of the Function object
  - It is created when a function is defined
  - When a function is used as a constructor, it indicates the prototype of objects constructed by said function
- **.__proto__** is an instance property of an object
  - References its prototype
  - This is the *prototype chain* we referred to earlier

Prototypes continued…

- All objects inherit from `Object.prototype`
  - The Function object inherits from `Function.prototype`
- Prototype is like a fallback object of properties that an object can use when the property does not exist on the object itself
- When you request a property of an object, it checks the object, then its prototype, then the prototype's prototype, and so on…
  - The prototype chain
- This is the basis of inheritance in JavaScript

Pseudo-classical Inheritance

- Basically…
  - Create a Constructor Function
    - ```
      function Gizmo(id) {
          this.id = id;
      }
      ```
  - Add methods and properties that will be inherited
    - ```
      Gizmo.prototype.toString = function() {
          return “gizmo “ + this.id;
      }
      ```
  - Allow inheritance via prototype
    - ```
      function Hoozit(id) {this.id=id;}
      Hoozit.prototype = new Gizmo();
      Hoozit.prototype.test = function(){…}
      ```
      ```

Prototypal Inheritance

- There are no “classes”, only objects
- All objects “inherit” methods and properties from other objects
- Objects are treated as dynamic templates
- How to do it
  - `Object.create(prototype); // ES5`
- Prototypal inheritance is arguably “better”
  - Allows for multi-inheritance (mixins)
  - Considerably simpler than classical patterns

Object.create()

- Basis of “real” prototypal inheritance
- Creates an object with a specified prototype and optional properties
- `Object.create(prototype, properties);`
- What it is:

```
function object(o) {
  function F(){};
  F.prototype = o;
  return new F();
}
```

- Car example
  - http://jsfiddle.net/mrmorris/f0tsos80/
- Advanced examples (if we want)
  - http://jsfiddle.net/mrmorris/uc3k80k1/

introspection

- instanceof
  - Tells you whether an object was derived from a specific constructor
  - `anObj instanceof FooConstructor`
- `Object.isPrototypeOf`
  - In the prototype chain of *a*, does Foo.prototype appear
  - `Foo.isPrototypeOf(anObj)`
- `Object.getPrototoypeOf()`
  - Returns the actual prototype (__proto__)
  - Must be an object in ES5, ES6 correctly returns String.prototype for “astring”

Objects and prototype recap

- Objects are dynamic
- Inheritance behaves like delegation/linking
  - You can use an object as a template for another object ad infinitum
- Object properties (and methods) are looked for up the **prototype chain**
- **__proto__** is a special property linking an object to its prototype
- **prototype** is a property you set on constructors
- **this** in an object method references the object
  - But it can change, depending on how you call the function

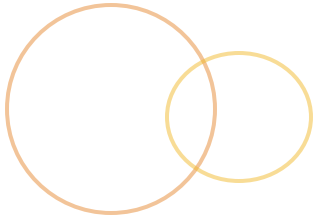
Delegation > inheritance

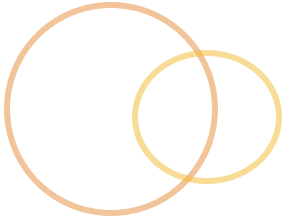
- JavaScript inheritance is more like “delegation” rather than classical inheritance which relies on copies
- Really is no need to use “new” or constructor functions
  - var foo = {…};
    var bar = Object.create(foo);
    var b1 = Object.create(bar);
    var b2 = Object.create(bar);

Exercise: Inherit something

- Using the “native constructor” technique, create a series of objects that inherit from their predecessors
- Choose a subject where there are plenty of types to pick from:
  - Animal -> Mammal -> Horse -> Clydesdale
  - Vehicle -> Automobile -> SUV -> Jeep
  - Profession-> Engineering -> Software Architect
  - Etc.
- No need to make several of each tier, just one inheritance chain is enough
- Feel free to add properties and methods that seem appropriate
- All done? Now try the same with Object.create() (instead of constructors)

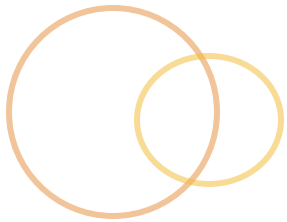
Day 2





- Any remaining set up issues?
- Any questions from day 1?
- Any comments/suggestions?
- For today:
  - JavaScript Core: Objects and Prototypes
  - HTML, CSS refresher
  - The DOM
  - Events
  - Native Ajax
  - jQuery basics

Warm-up Exercise



- Create an object that represents yourself
  - Give it a name (ex: Ryan)
  - A height? Age? Whatever
- You should be able to do stuff, like speak
  - Create a function on your object, “speak”, accepts a string argument and logs it to the console as:
    - “Ryan says: {the string}”
- Let’s assume you also have a trophy collection
  - Create an array property on your object that has a couple trophies inside it
  - Add a function that lets you view trophies:
    - me.viewTrophy(i); // will log the name of the trophy at i index
  - Add a function that lists all trophies, separated by commas
    - me.listTrophies(); // -> “gold star, track, silver medal”
- How might we make it so that your trophies are protected from theft?
  - Hint: IFFE, Closures, and entering into modules…

Day 2: JavaScript & the DOM

- HTML/CSS recap
- The DOM
  - Content creation and traversal
  - Events
  - Ajax

Debugging in the browser

- $_; // the value of the last evaluation
- $0 through $4, last inspected elements
- $("selector"); // first matching node
- $$("selector"); // all matching nodes
- debug(function); // sets a breakpoint at function
- monitor(function); // trace calls to a function

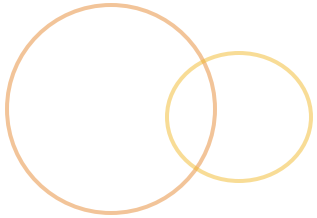
JavaScript and the Browser

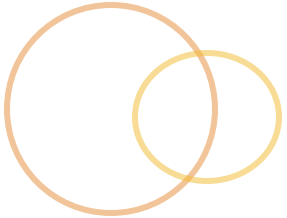
- Why JavaScript
  - Interactivity, based on user or browser triggered events
  - Load data into the page dynamically
  - Built in business logic
  - Single page applications (if that's your cup of tea)
  - The web is the new application platform…
- How it fits
  - HTML for view data & ui structure
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

Wizard 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?

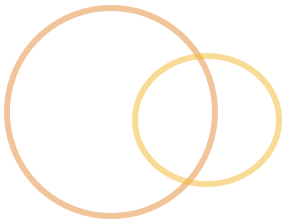
HTML



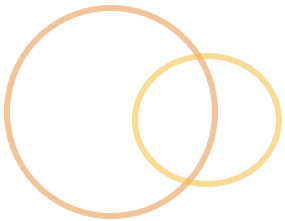
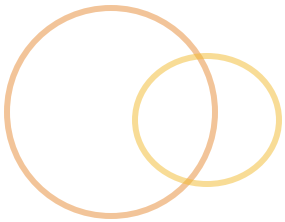
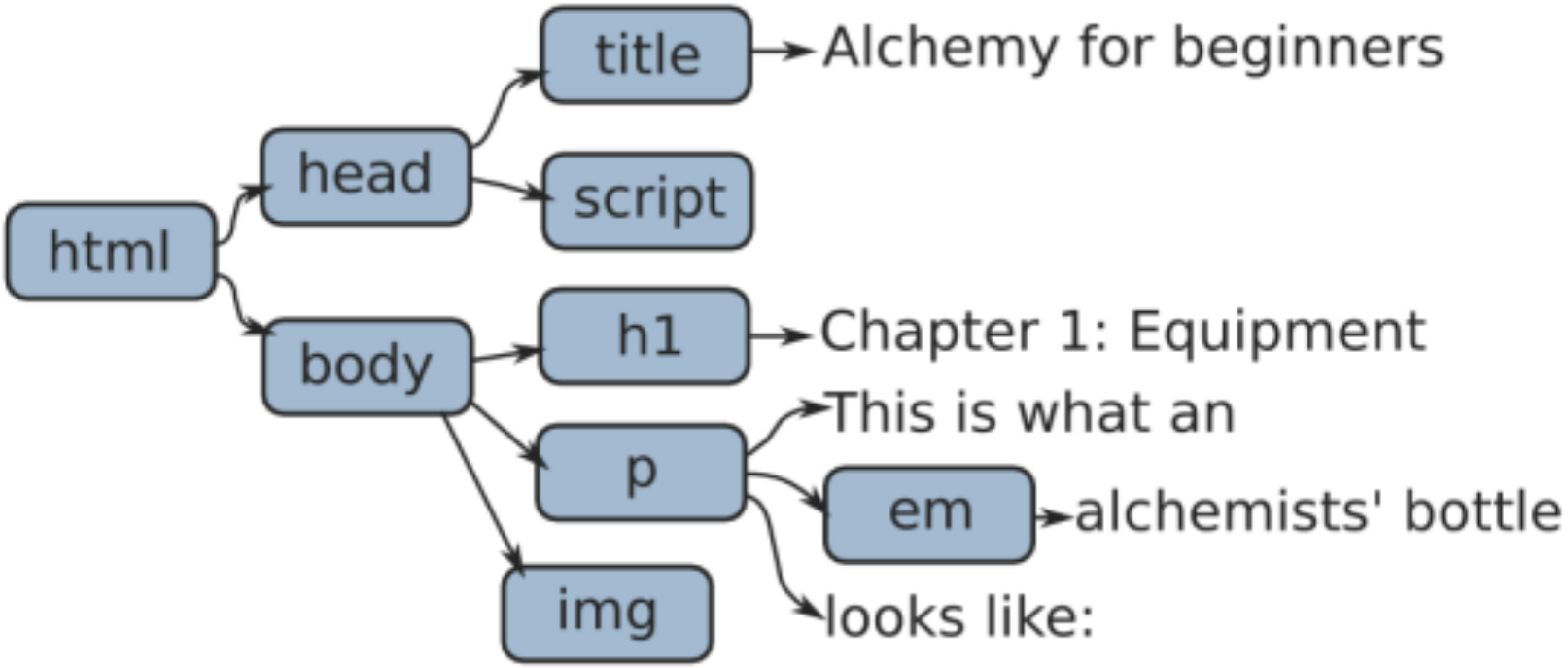
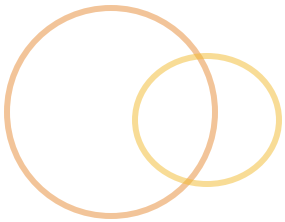
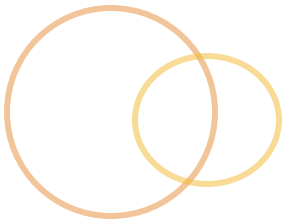
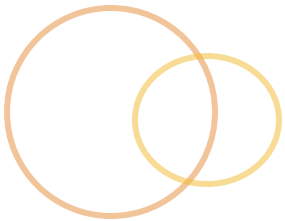
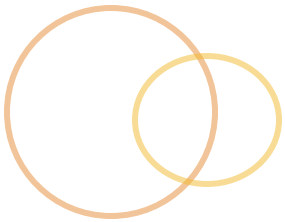
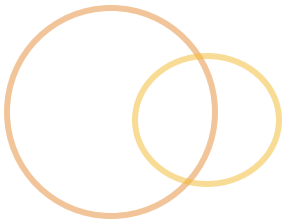
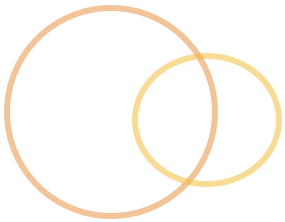
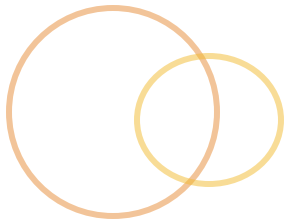


- Hyper Text Markup Language
- Browsers allow support for all sorts of errors – html is very error tolerant
- Structure of the UI and “view data”
- Tree of element nodes
- HTML5
  - Rich feature set
  - Semantic
  - Cross-device compatibility
  - Easier!

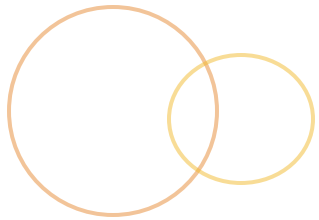
Anatomy of an html page

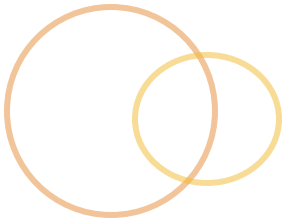
- The document
  - ```
    <!doctype html>
    <html lang="en">
     <head>
          <meta charset="utf-8">
          …document info and includes…
     </head>
     <body>
          <h1>Hello World!</h1>
     </body>
    </html>
    ```
- http://jsbin.com/retupe/edit?html
    ```

HTML as a tree...



Anatomy of an html element

- Aka nodes, elements, tags
- <element attributeName="attributeValue">
  Content of element
  </element>
- Block vs inline
  -
  -
- Self closing elements
  - <input type="text" name="username" />

HTML Elements refresher

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

CSS

- Cascading Style Sheets
- 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

```
selectors {
    /* 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 {}
- By class
  - .main {}
- By attribute
  - div[name="user"] {}
- By relationship to other elements
  - li:nth-child(2) {} ***
***

CSS Specificity

- Multiple selectors will override each other based on “specificity”
  - ```

     What color will I be?

    ```
  - ```
    #main{
     color: #f90;
    }
    .fancy{
       color:blue;
    }
    #main.fancy{
     color:red;
    }
    ```
- Order of specificity: inline, id, psuedo-classes, attributes, class, type, universal
- !important

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 (Performance)

- 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



- My text
- Maps to: {

```
    …
    childNodes: NodeList[1],
    className: "hi",
    innerHTML: "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 – Selecting

- Start on `document` or a previously selected element (except getElementById)
- Returns a `NodeList`
  - `el.getElementsByTagName("a");`
  - `el.getElementsByClassName("className");`
  - `el.querySelectorAll("css selector");`
- Returns a **single element**
  - `document.getElementById("idname");`
  - `el.querySelector("css selector");`
- http://jsfiddle.net/mrmorris/wcff257b/

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/mrmorris/dv13y28m/

Traversal ES5+



- .children
- .nextElementSibling
- .prevElementSibling
- .firstElementChild
- .lastElementChild

- These avoids annoying text-node stuff

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 – Creating

- http://jsfiddle.net/mrmorris/ktwdye0w/
- `document.createElement()`
  - Creates and returns a new node without inserting it into the document
- `document.createTextNode()`
  - Creates and returns a new text node
- `el.innerHTML`
  - Can set the inner html content of a node

DOM – Setting Content

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

DOM - insertion

- On an element or the document
  - `.appendChild(node); // appends to childNodes`
  - `.removeChild(node); // removes from`
  - `parentEl.insertBefore(newEl, beforeEl);`
  - `el.replaceChild(newNode, oldNode)`
    - oldNode must be a child of “el”

DOM – Element Attributes

- http://jsfiddle.net/mrmorris/duopdjdb/
- 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/mrmorris/hJwCj/

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`
- `.getBoundingClientRect()`
  - Returns object with top, bottom, left, right properties relative to top left of the page
- If you want it relative to document, add on current scroll position (globals)
  - `pageXOffset, pageYOffset`

DOM Performance



- Dealing w/ browser 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()

Exercise: DOM Manipulation

- ~~Start the Node.js server~~

  ~~`node bin/server.js`~~
- Open the following file:

  `www/flags/flags.js`
- Complete the exercise
- Run the tests by visiting in your browser:

  `http://localhost:3000/flags/`

Exercises: Dom manipulation

- 1. 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/mrmorris/97ukrors/
- 2. 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 
    }
    ```
    ```

Events





- JavaScript engine has an event-driven, single-threaded, asynchronous programming model
- As things happen
  - User clicks
  - Page completes loading
  - Form is submitting
- 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 to
- 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/mrmorris/YAnBV/

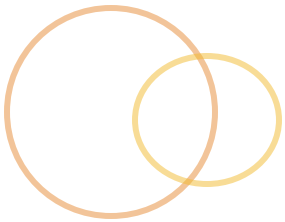
Context… in event handling

- 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 “bind”

Event handlers with arguments

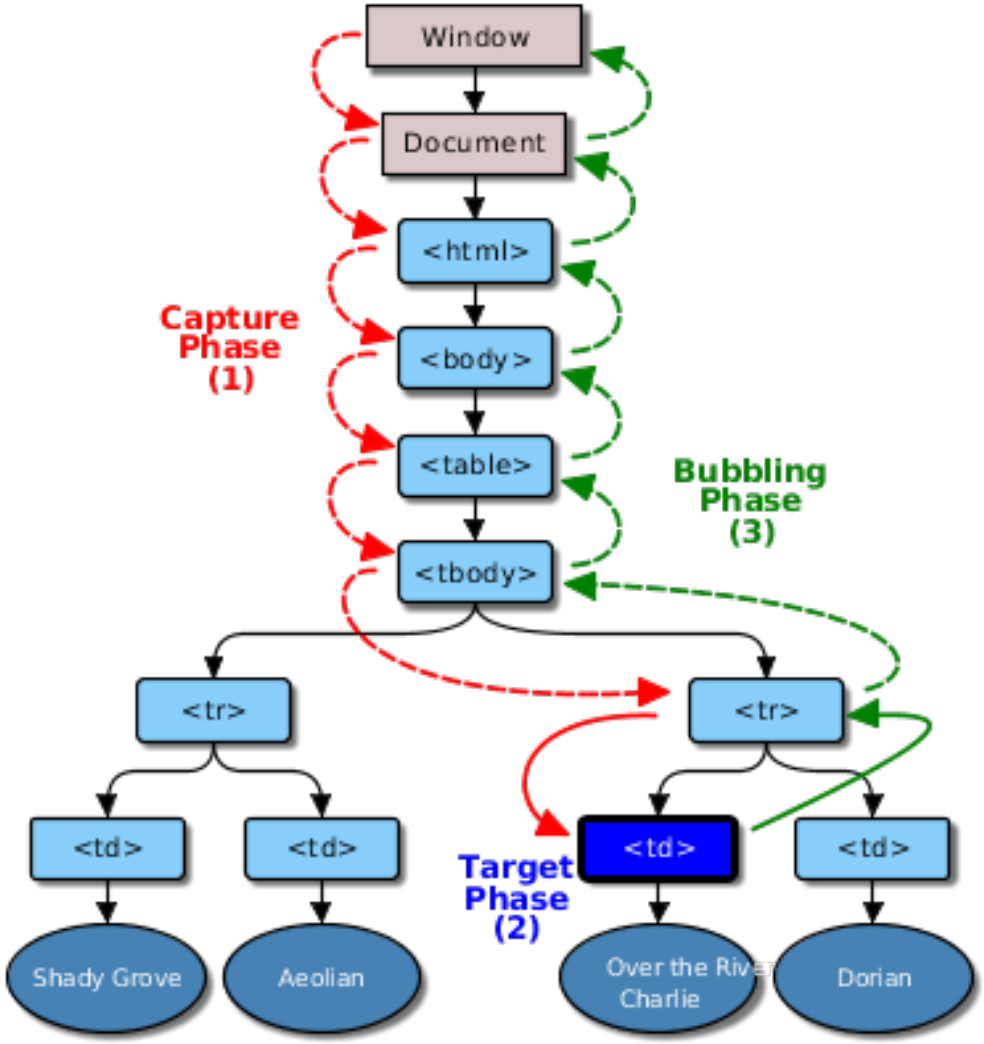
- This won't work like you expect
  - element.addEventListener('blur', doSomething(5));
- It is passing the result of a function invokation, not the function to-be-called as a callback
- Instead, wrap it in a function
  - el.addEventListener('blur', function() {
    doSomething(5); // func needed an arg
    });
- Or use Function**.bind**
  - el.addEventListener('blur', doSomething.bind(el, 5));

Event Propagation

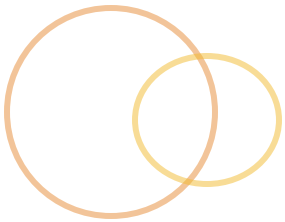
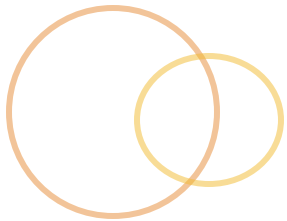


- An event triggered on an element *elA* is also triggered on all parent elements of *elA*
- Two models
  - Trickling, aka Capturing phase (Netscape)
  - Bubbling (MS)
- W3C decided to support both
  - Starts in capturing, then bubbling
  - Defaults to bubbling
- Bubbling supports Event Delegation
 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

Event Propagation, continued



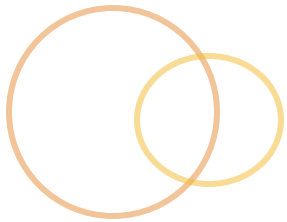
Stopping events

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

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

Event information



- 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/mrmorris/8htsexcg/

Complete example



```
.addEventListener('click', function(event) {

    // "this" represents the element handling the event
    this.style.color: "#ff9900";

    // "target" represents the element that triggered
    event.target.style.color: "#ff9900";

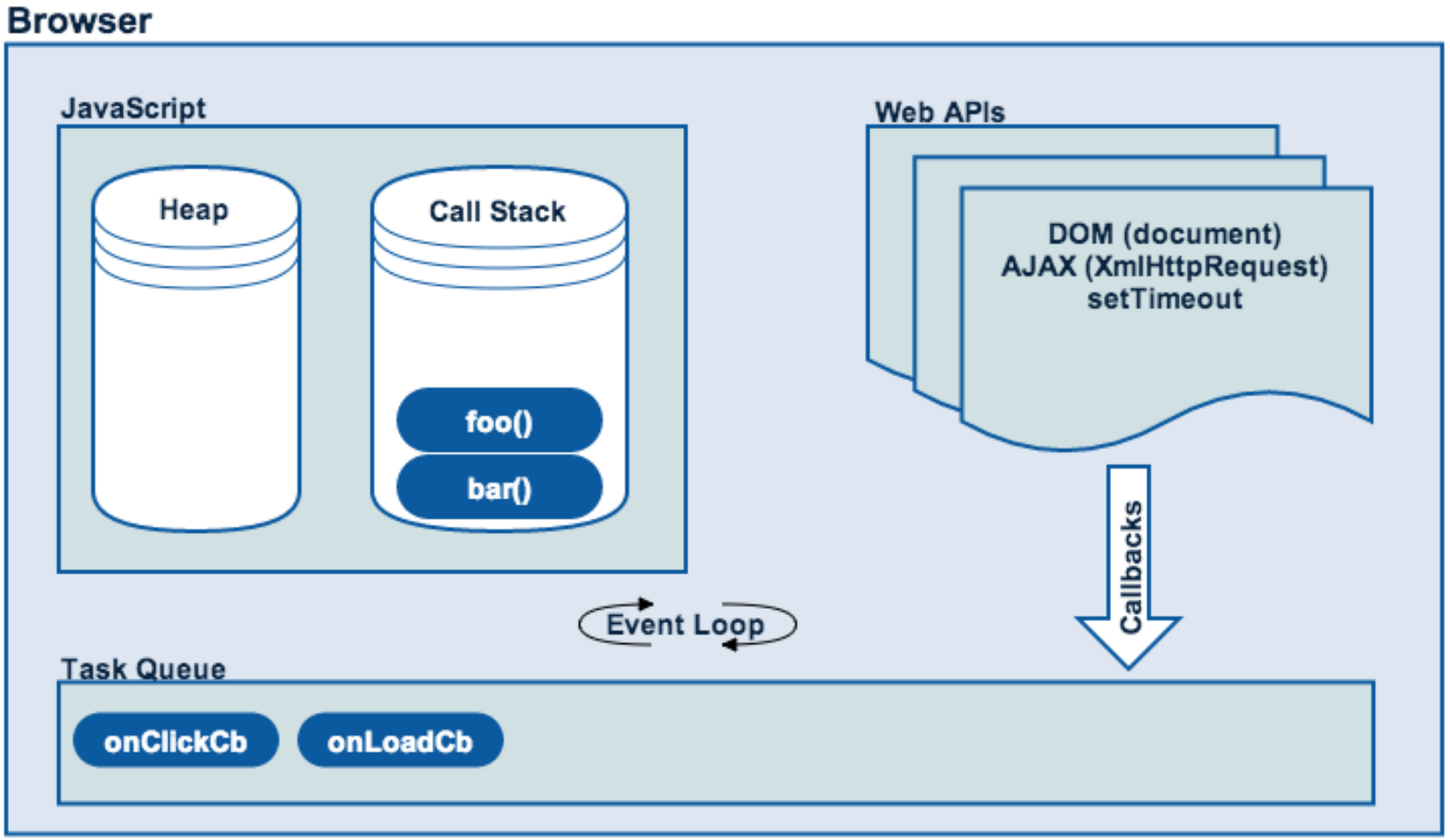
    // you can stop default browser behavior
    event.preventDefault();

    // or you can stop the event from bubbling
    event.stopPropagation();
});
```

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/mrmorris/2gK7L/

Event debouncing

- “debounce” events that fire rapidly like `mousemove`, scroll
  - ```
    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/mrmorris/a7GTG/
    ```

Event Loop

- Heart of the browser's JavaScript engine (one of the best parts)
- Allows asynchronous operations
  - Methods that get tacked into the queue are 'async'
- Each "turn" it returns a function from the Queue and runs it to completion (blocking)
  - Avoid blocking scripts…
    - alert, confirm, prompt, synchronous xhr/ajax
- For long running tasks…
  - *Eteration*: break task into multiple turns and call each with a setTimeout
  - Move the task into a separate process

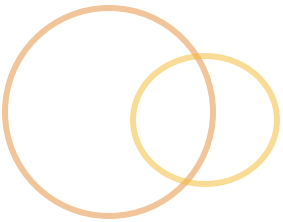
The event loop



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
- `setTimeout(0)` will push the task to the end of the main program by putting it in the task queue. Then the main program goes through the stack.

Event Examples



- Style a form field when prefilled
  - http://jsfiddle.net/mrmorris/t1o9ptpL/

Events recap

- Events are notifications that bubble up from different sources in the page
  - Usually through a user doing something
  - Or some content in the page doing something
- Event delegation allows you to register a single handler to handle many (child) nodes' events
- The browser event loop is powerful but it is single-threaded so a long-running process can halt all interactions in the page.

Exercise: DOM Manipulation

- ~~Start the Node.js server~~

  ~~`node bin/server.js`~~

- Open the following file:

  `www/flags/events.js`

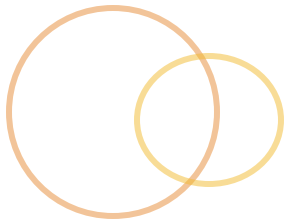
- Complete the exercise
- Run the tests by visiting in your browser:

  `http://localhost:3000/events/`

AJAX/XHR

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

XHR– Step by step



1. Browser makes a request to a server
2. Script continues along it's merry way
3. Server responds in xml/json/html
4. Browser parses and processes response
5. Browser invokes our JavaScript callback

XHR - Making a request

- Create a request object and call its “open” and “send” methods
  - `var req = new XMLHttpRequest();`

```
// attach listener

req.open(“GET”, “url.json”, false);
req.send(null);
```

XHR– Handle the response

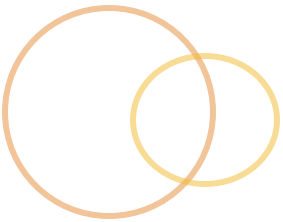
- “load” event will fire when response is received

```
req.addEventListener(“load”, function(e) {
    if (req.status==200) {
          console.log(req.responseText);
    }
});
```

Data formats

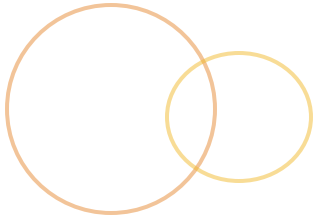
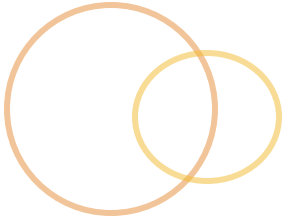
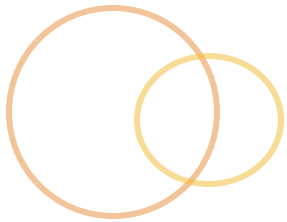
Format	Summary	PROS	CONS
HTML	Easiest for content in page	• Easy to parse • No need to process much	• Server must produce the HTML • Data portability is limited • Limited to same domain
XML	Looks similar to HTML, more strict	• Flexible and can handle complex structure • Processed using the DOM	• Very verbose, lots of data • Lots of code needed to process result • Same domain only
JSON	Similar object literal syntax	• concise! Small • Easy to use within JavaScript • Any domain, w/ JSONP or CORS	• Syntax is strict • 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
- An example:
  - http://js-jquery.course/ajax/index-html.html

XHR with XML

- More work in processing the data to turn XML into HTML
  - ```
    var data = xhr.responseXML;
    var events =
    data.getElementsByTagName('event');

    for (var i=0; i<events.length; i++) {
     var container = document.createElement('div');
     container.className = 'event';
     // create img node
     // append
    }
    ```
- http://js-jquery.course/ajax/index-xml.html
    ```

JSON

- **J**ava**S**cript **O**bject **N**otation
- Most commonly used web data communication format
- Like an object literal, except:
  - Property names must be surrounded by double quotes
  - No function definitions, function calls or variables
- Methods
  - `JSON.stringify(object);`
  - `JSON.parse(string);`
- http://js-jquery.course/ajax/data/data.json

JSON






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

XHRwith JSON



- It is sent and received as a string and will need to be de-serialized
  - ```
    var data = JSON.parse(xhr.responseText);
    var newContent = “”;

    for (var i=0; i< data.length; i++) {
     newContent += ‘ ’;
     newContent += ‘<img src=“’ + data[i].val+
    ‘”/>’;
    }

    document.getElementById(‘content’).innerHTML
    = newContent;
    ```
- http://js-jquery.course/ajax/index-json.html
    ```

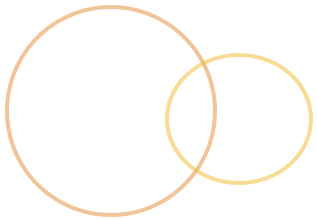
XHR usage

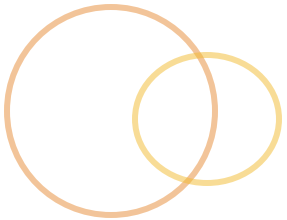
- It is best to use an abstraction of `XMLHttpRequest` for
  - `status` and `statusCode` handling
  - Error handling
  - Callback registration (`onreadystatechange` vs `onload`)
  - Browser variations and fallbacks
  - Additional event handling
    - `progress`
    - `load`
    - `error`
    - `abort`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

Cross-origin

- By default, ajax requests must be made on the same domain
- Alternatives to this are:
  - A proxy file on the server
  - JSON/p “Json with padding”
  - CORS (Cross-origin resource sharing), which involves new http headers between browser and server – ie10+
- For later: http://jsonplaceholder.typicode.com/

JSONp





- Browsers don't enforce same-origin policy on the `src` in script tags
- So…
  - We define a handling **callback** function
  - We dynamically add a **script** referencing an external `src`
  - 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
- The callback function is expected to be defined on the origin
- http://js-jquery.course/ajax/jsonp-example.html

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)

Exercise: Making Ajax Requests

- ~~Start the Node.js server~~

  ~~`node bin/server.js`~~

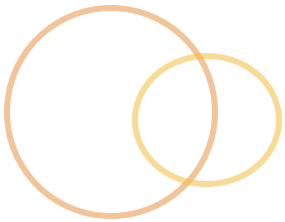
- Open the following file:

  `www/ajax/ajax.js`

- Complete the exercise
- Run the tests by visiting in your browser:

  `http://localhost:3000/ajax/`

jQuery Ajax review



- ```
  $.ajax({
      type: 'GET', // or 'POST', 'DELETE',
      data: {},
      success: callback
      error: callback
      complete: callback
      dataType: 'json', // 'json','html'
  });
  ```
- $.ajax (and shortcuts) method immediately (synchronously) returns a Promise object
  - ```
    var prom = $.ajax({…});
    prom.done(function(response){…});
    prom.fail(function(){…});
    prom.always(function(){…});
    ```
- These promise methods can be chained
  - `prom.done().fail().always()`

Exercise: JavaScript and the DOM

- 1) Create a tabbed UI that responds to click events and adding new tabs.
  - http://jsfiddle.net/mrmorris/osq6fed3/
- 2) Create a table-builder function that accepts JSON data and builds a table element with all the trimmings.
  - http://jsfiddle.net/mrmorris/mnyn3y0t/

Stay sharp

- Solve small challenges for kata
  - http://www.codewars.com/
- Code interactively
  - http://www.codecademy.com/
- Share your code and get feedback
  - http://jsfiddle.net
- Free e-book
  - http://eloquentjavascript.net/
- Re-introduction to JavaScript
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

Team exercise

- Let's make a jQuery plugin
  - What are some things you want to achieve with jQuery?
  - What are some reusable behaviors or components you wish you had available?
  - Is there some behavior you want in reaction to a user's interaction?
  - A backup idea:
    - Ability for a form to be aware of when a field is "pristine", "dirty", "changed" and "unchanged"
      - Pristine: untouched field value
      - Dirty: touched field value
      - Changed: changed from original
      - Unchanged: unchanged from original

Go now and code well

- That's a wrap!
  - What did you enjoy learning about the most?
  - What is your key takeaway?
  - What do you wish we did differently?
- Any other comments, questions, suggestions?
- Feel free to contact me at **mr.morris@gmail.com** or my eerily silent twitter **@mrmorris**