learning spike

Core JavaScript

Ryan Morris

@mrmorris

Introductions

- About me…
- About you…
  - Name?
  - What do you do here?
  - What is your programming background?
    - Any front-end?
    - 😍, 😡 or 😭 JavaScript?
  - What do you hope to gain from this course?

How the class works

- Lecture & labs
- Informal
- Flexible outline
  - You help me define areas of interest
  - Too much to cover!
- Exposure to *Core JS* concepts
- Class review at the end of the day

Get the most out of the class

- Ask **questions**!
- Do the **labs** (pair up if needed)
- Be **punctual**
- **Avoid distractions**
- Master your **google-fu**
- **Play along** in the console
- Don't be afraid to **break stuff**

What we'll cover

- Syntax basics
- Coercion
- Scope
- Hoisting
- Objects (basics)
- Functions (basics)
- Context
- Closures?

I wasn't planning to cover

- Objects in depth
- OO/inheritance
- ES6
- Modules

~Mostly ES5~

~Mostly for beginners~

Resources

- Reading List
  - https://javascript.info/intro
  - You Don’t Know JS
    - https://github.com/getify/You-Dont-Know-JS
- Documentation
  - http://devdocs.io
  - https://developer.mozilla.org/en-US/docs/Web
  - Google it.
- Compatibility checks
  - http://caniuse.com

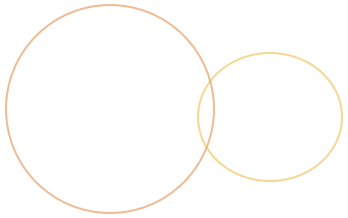
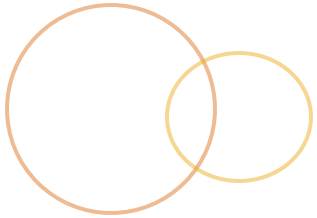
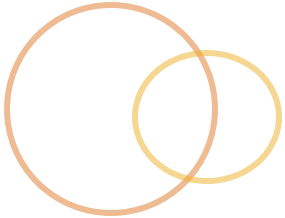
Set up

- A browser with dev tools
  - Preference for Chrome in class
  - Open your browser and hit `F12` or `alt/opt/⌥ -⌘-i`
- Sign up with jsFiddle.net
  - http://jsfiddle.net/
  - Does this work?
    - http://jsfiddle.net/mrmorris/8wfu5tct/
    - You should see “We are ok!” message
      - No? Are you in http or https?
- Slides:
  - https://github.com/rm-training/resources/blob/master/spikes/spike-core-js-rmorris-2018.pdf

👆 Everyone OK with the above?

jsFiddle primer

- It's a **sandbox**
- It's a set of **iframes**
  - Check which frame you're accessing via your console
- It runs in an **IIFE** unless you ask it not to
  - So your stuff isn't global…
- When you start a lab…
  - Fork it (copy) — you'll own that!
  - "*update*" to save!
  - "*run*" to test!
  - "*set as base*" to make a version the *main* version


module

JAVASCRIPT INTRO

History

- “Make webpages alive”
- 1995 - Netscape wanted interactivity like HyperCard w/ Java in the name
- Designed & built in 10 days by Brendan Eich
  - initially named "Mocha", released as “LiveScript"
  - Became “JavaScript” once name could be licensed from Sun
- Combines influences from:
  - Java, "Because people like it"
  - SmallTalk, prototypal

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)
- Safe (no CPU or memory access)
- Depends on the engine + environment running it
- *Kind of weird but enjoyable*

JavaScript Versions

- 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 [EcmaScript 2015] mostly supported**
- ES7 [EcmaScript 2016] finalized, but weak support
- ES8 [EcmaScript 2017] finalized in June 2017
- ES.Next…

Why JavaScript?

- Scrappy, flexible and powerful
- The language of the web
  - Integrates nicely w/ HTML/CSS
  - Supported across all browsers
- Beginning to dominate the entire stack
- *Easy to learn, hard to master*

Approaching JavaScript

- It's *not* Java
- It's *not* class-based
- Very dynamic & flexible
- Supports many paradigms
  - imperative, functional and object-oriented
- Be aware of the downsides
  - Single-thread/Blocking
  - Evolved w/out ever cleaning the closet
  - Lot’s of parties involved in its evolution
  - Flexibility requires understanding

Where does JavaScript live?

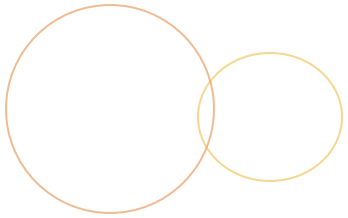
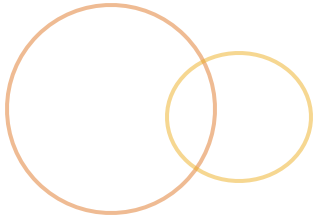
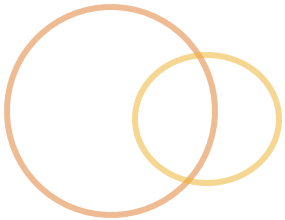
- Plain text files, not compiled
  - *Though this is changing*
- In your browser (Built-in Engine)

```
// external script files
<script src="app.js"></script>
// or inline block
<script>
  alert('Hello World!');
</script>
```

- On your server (Node)
  - One or more scripts and modules

Not quite JavaScript

- JS doesn’t always meet everyone’s needs
- Transpile (compile) down to plain JavaScript
  - CoffeeScript - syntax sugar
  - TypeScript - strict data typing
  - Dart - non-browser environments
  - ClosureCompiler
  - and more!


obligatory

HELLO WORLD

Alert hello

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

```
alert('Hello World!');
```

- Alternatively…
  - In a <script>
  - or… in a file linked from an HTML page
  - or… run by NodeJS

Log hello

- Now try

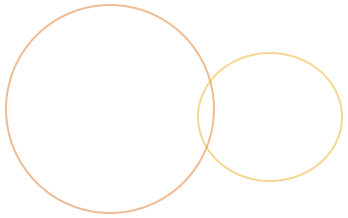
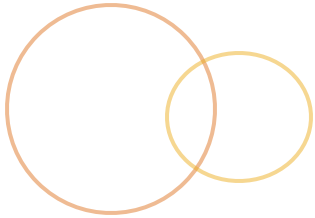
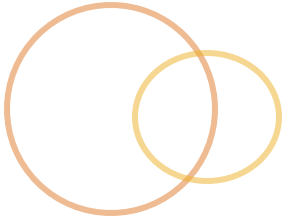
```
console.log('Hello Engineers!');
```

Browser Debugging

- Use browser dev tools to access its JavaScript console
  - The browser's "`console`" is a REPL
  - log output for testing
- Can also use dev tools to:
  - set breakpoints & debug js
  - view network requests
  - view memory usage
  - inspect html + css

The `console` object

- Console api
  - `console.log(); // echo/print/output`
  - `console.assert(); // test`
  - `debugger; // breakpoint`
- Gotchas
  - Console methods are asynchronous
    - They may not run in the order you expect
  - They are not available in every browser
- Seeing a bug/issue?
  - Clear your console of old errors
  - Check where the error happened


module

SYNTAX BASICS

Syntax Examples

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

C-family syntax

- Instructions are **statements** separated by **semi-colon**

```
var x =5; var y = 7;
```

  - Spaces, tabs and newlines are **whitespace**.
  - White space and indentation generally doesn't matter
- **Blocks** are wrapped with curly braces

```
var x = 5;
if (x) {
  x++;
}
```

```
{
  x = 5;
  y = 7;
}
```

Automatic Semicolon Insertion

- Semicolons terminate statements

```
y = 5 + 1;
```

- They are *mostly* optional
  - Automatically inserted but not fail-safe
  - So, don't rely on it…

```
var fn = function() {
  // do stuff
}
(function() {
  // do stuff
})();
```

Missing semi-colon here results in a `TypeError`

Comments

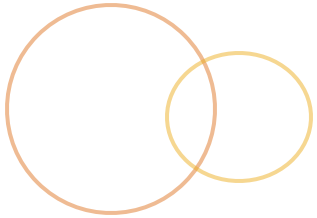
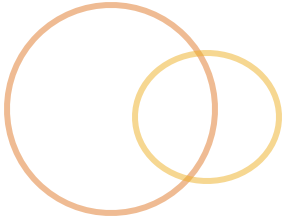
- Follow C/C++ conventions
- Multiline

```
/*
span multiple
lines
*/
```

- Single line

```
// I can comment one line at a time
var x = 1; // wherever
// var x = 5;
```

Variables

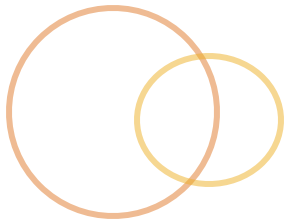





- Used to store or reference a value

```
var MyName;
var your_name;
var $; // like jQuery
var _myName;
var num10;
var 🍔 = 'burger';
```

- Var names can contain **letters**, **digits**, _, or **$**
  - But can't begin with a digit
  - No reserved keywords
  - ***CaSE*** matters
  - **Unicode** characters are supported

Declaring variables



- With the keyword **`var`**
  - and **`let`** or **`const`** in ES6+
- One by one:

```
var foo;
var thing1;
```

- Or in sequence:

```
var a, b;
```

- Default value will be `undefined`

```
var another;
console.log(another); // “undefined”
```

Assigning values

- Use = to assign values to variables

```
var x = 5;
var y = 1, z = 'rad';
```

- Can assign and re-assign at any time

```
var x;
x = 10;
x = false; // ok!
```

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

```
stuff = [1,2,3];
stuff; // [1,2,3];
window.stuff; // [1,2,3];
```

Data Types

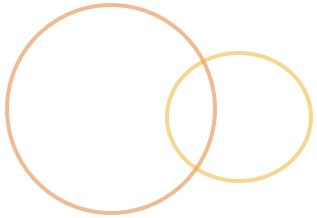
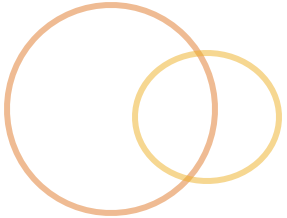
- Five *primitive* data types:
  - null - *lack of value*
  - undefined – *no value set* (default)
  - strings
  - numbers
  - booleans
  - *ES6: Additional primitive, Symbol*
- And then *Objects*
  - Property names referencing values
  - ie: `Object`, `Array`, `Function`, `Math`…
  - Function is a callable object
  - All *primitives* have *Object* counterparts

`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`**

```
var isRyanTall = true;
var do_something = false;

if (isRyanTall) {
  // do something…
}
```

number

- 64bit floating point
- Numbers can be expressed as:
  - Decimal: **-9.81**
  - Scientific: **2.998e8**
  - Hexadecimal: **0xFF;** // 255
  - Octal: **0777**
  - Binary: **0b10000000000000000000000000000000**

```
var x = 1;
var y = 1.5;
var z = -3;
```

Number Issues

- Maximum number length (up to 15 digits)

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

- Decimal inaccuracies (up to 17th place)

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

- NaN

```
0/0; // NaN
NaN == NaN; // false ?? gotcha…
```

- Infinity and -Infinity

```
5/0 // Infinity
```

string

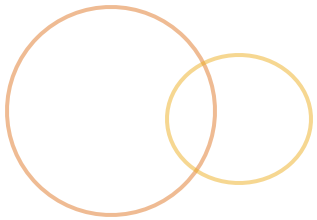
- Enclosed by " or ' (just don't mix them)

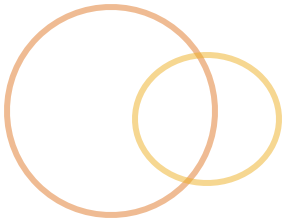
```
var str = "My Name Is”;
var name = 'Ryan';
```

- Escape with backslash (\)
  - **\n** is newline, **\t** is tab, etc
- Concatenate with + operator

```
"Hi, “ + str + " " + name + "!";
```

Objects





- A list of **key:value** pairs, surrounded by **curly braces**
  - Considered a *Dictionary*, *Hash* or *Map* in other languages

```
var dog = {
  name: “fido”,
  age: 12
};
dog.hasTail = true; // assign values
dog.name;   // dot-accessor
dog[‘name’]; // array-accessor
```

- Keys:
  - unordered
  - Strings, quotes only required if they include special chars
- Values:
  - any type of data, including functions

Objects, continued…

```
var person = {
  name: 'Ryan',
  isTall: true,
  speak: function() {
    console.log('Hi');
  }
}

person.name; // Ryan
person.speak(); // Hi
```

Functions (intro)

- Functions are **callable objects**

```
function sayHelloTo(name) {
  console.log('Hello ' + name);
  return name + "!";
}
sayHello('Ryan'); // Hello Ryan
```

- They can be referenced by a **name** or **variable**
- They can exist on objects as **methods**
- They can expect **arguments**

Arrays

- Data stored *sequentially* with an index

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

- In JavaScript, an **array is an object** that behaves *kinda* like an array (*array-like*)
- Strange behavior if you try to use string keys

```
var arr = [1,2,3];
arr.length; // 3 <- three items
arr['bar'] = 10;
arr.length; // 3 <- hmm i expected 4?
```

array methods

- Arrays are objects…
  - … and have additional properties and methods

```
myArray.length; // 4
myArray.push('John'); // adds value to end
myArray.pop(); // John
```

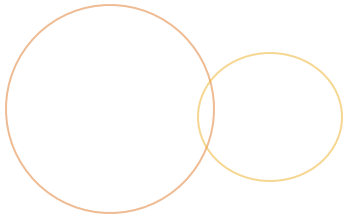
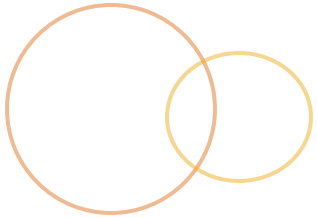
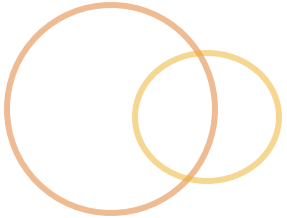
- In fact, ***everything*** can act like an object…
  - … and has additional properties and methods

```
var name= "John Smith";
name.length; // 10
"foo".toUpperCase(); // "FOO"
5..toString(); // 5
```

Exercise – Data Types

- Experiment directly in your console
- Quick review of jsfiddle…
- **Super Basic Data**
  With built-in tests
  - Fork this:
    - https://jsfiddle.net/mrmorris/gzpo0z0L/

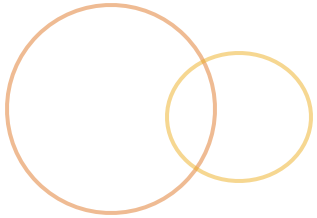
Solutions:
Super Basic Data: https://jsfiddle.net/mrmorris/5gdv03r0/

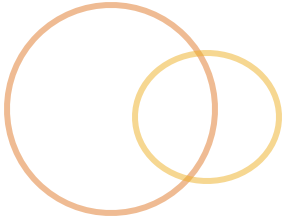





module

SYNTAX - OPERATORS

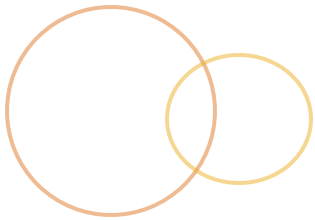
Unary

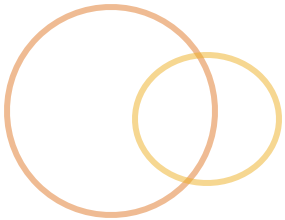




```
delete obj.x    // undefined
void 5 + 5      // undefined
typeof 5        // 'number'
+'5'            // 5
-x              // -5
~9              // -10 (bitwise flip bit)
!true           // false
++x             // 6
x++             // 5
--x             // 4
x--             // 5
```

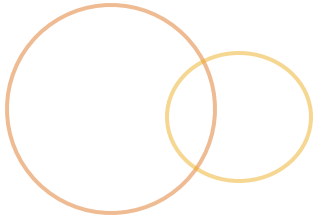
Arithmetic

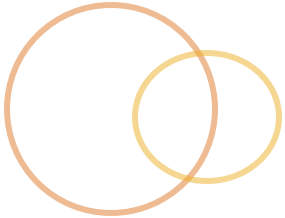




```
5 + 5       // 10
5 - 3       // 2
5 * 2       // 10
10 / 2      // 5
10 % 3      // 1
```

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
// &=,|=,^=,<<=,>>=,>>>=
```

Getting the type of a variable

- **typeof** returns the type of the argument

```
typeof undefined; // “undefined”
typeof 0;         // “number”
typeof “foo”;     // “string”
typeof true;      // “boolean”
typeof null;      // “object” ???
typeof {};        // “object”

// can use as a function
typeof(0);        // “number”
```

typeof objects

- **typeof** with any* object is **“object”**

```
typeof {};       // “object”
typeof [1,2,3]; // “object”
typeof Math;     // “object”
```

- *except **Functions**

```
typeof alert;     // “function”
```

Exercise - typeof an Array?

```
var myArray = [1,2,3];
typeof myArray; // ?
```

Everything* is an object

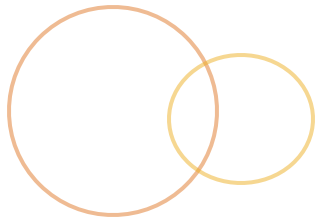
- Primitive literals all have Object counterparts
  - except null and undefined

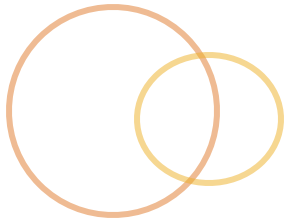
```
5 === Number(“5"); // true
"Hello" === String(“Hello"); // true
true === Boolean(1); // true
```

- *most things, primitives are just ***coerced***
- So… we can access properties and methods of objects, including primitives

```
var str = “bla”;
str.length; // 3
str.toUpperCase(); // BLA
“Hello”.length; // 5
```

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
```

Don’t construct your literals

- Because they have object counterparts, one can ***construct*** them to create ***new instances***

```
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(); // {}
```

- ***But:***
  - Uses additional memory/cpu
  - Some side-effects
  - Too class-based

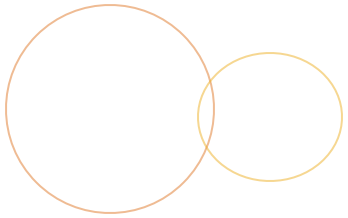
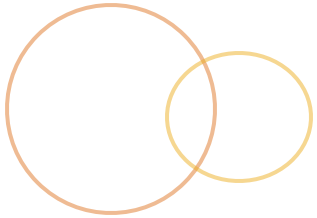
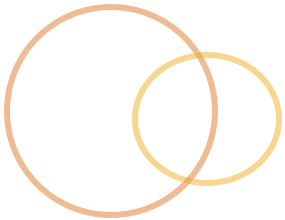
Recap: basic data types

- 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

Exercise - Syntax and Operators

- **Working with variables**
  Experiment with setting variables and manipulating their data. Wow!
  - Fork this:
    - http://jsfiddle.net/mrmorris/e5g7ub2n/

Solutions:
Working with Variables: http://jsfiddle.net/mrmorris/bayoa6ao/


module

CONTROL STRUCTURES

Conditionals & Loops

Control Structures & Logic

- We'll use control structures & logical expressions to define the flow of our script
  - `if` and `if-else` statements
  - `switch` statements
- And to process data
  - `for` and `while` loops to repeat actions or loop over arrays
  - what about objects?

Control Structure Examples

- http://jsfiddle.net/mrmorris/GN7qL/

Conditional statements

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

```
    …
} else {
    …
}
```

- `if {} else if {} else {}`

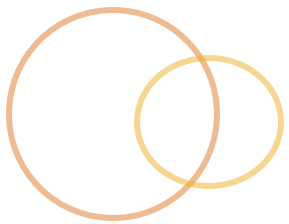
Relational operators

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

Equality operators (strict vs loose)

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

Logical operators

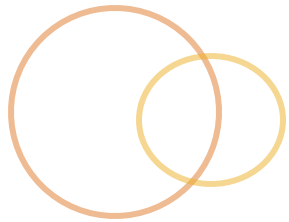


```
false && 'foo'   // false

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

Conditional example

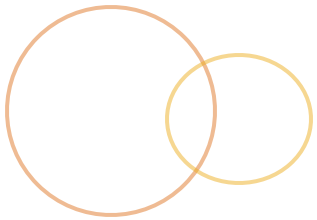
```
// generates a value between 0 and 1
var rand = Math.random();

if (rand > .1 && rand < .3) {
  // do something
} else if (rand === .4) {
  // do something
} else {
  // do something
}
```

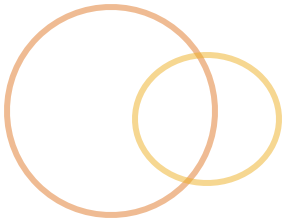
Switch statements



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

    default:
        // statements
        break;
}
```

Ternary





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

looping - for

- Do something {x} times

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

- Loop over an array

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

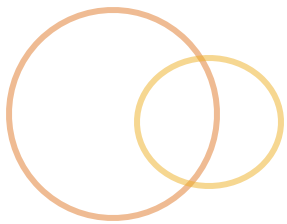
looping - while

```
var i = 0;


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

looping - do/while

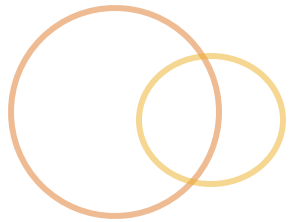
```
var i = 0;


do {
    // do at least once
    i++;
} while (i < 10);
```

Break and Continue



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

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;
}
```

Where short-circuits help

- Default function values

```
function name(x) {
  // set default value of x if undefined
   x = x || null;
}
```

- Gateways

```
return obj.name
  && obj.id
  && obj.doSomething();
```

Control Structures Recap

- Conditionals like if and if-else
- Switch statements
- Iterate (loop) with while and for

Exercise – Control flow

- **Control Flow 1**

  Get to know control flow and iteration statements

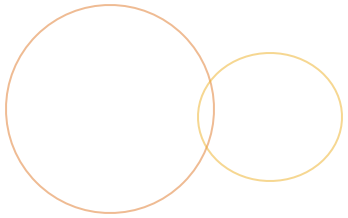
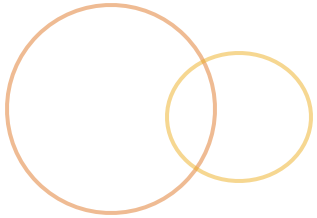
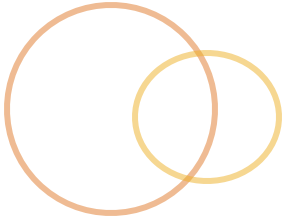
  - We'll use some basic browser functions
    - `alert("A message!");`
    - `var response = prompt("Ask for a value!");`
    - `confirm("Ask user to say 'ok'");`
  - Fork me: http://jsfiddle.net/mrmorris/cxz2hta1/

- **Control Flow 2**

  Control flow trials with built-in tests

  - Fork me https://jsfiddle.net/mrmorris/cvkgnuq3/

Solutions:
Control Flow 1: http://jsfiddle.net/mrmorris/1yb31dt6/
Control Flow 2: https://jsfiddle.net/mrmorris/nr1pwtcq/


module

COERCION

Type Coercion

- If a variable type is not what JavaScript expects, it will convert it on the fly, based upon the context
  - Just like a primitive is coerced to an Object

```
“ryan”.length; // coerced to a String()
```

  - In numeric expressions with the **+** operator, numbers may be coerced to strings (and vice versa)

```
+"42"; // 42
"Name: " + 42; // "Name: 42"
1 + "3"; // 4;
```

Implicit Coercion

- It's not obvious how it will coerce…

```
8 * null; // 0
“5” – 1;  // 4
“5” + 1;  // 51
```

- Much confusion ensues

```
[] + []; // “”
[] + {}; // [object Object]
{} + []; // 0
{} + {}; // NaN
```

Sometimes coercion is cool

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

typeof and NaN

```
typeof NaN;  // "number" <- huh?
NaN === NaN; // false  <- bummer…
isNaN(NaN);  // true
```

- `isNaN()`
  - is… "is this var NaN"… not "is this not a number"
  - *coerces*

```
isNaN(''); // false
isNaN(5); // false
isNaN('123ABC'); // true
isNaN({}); // true
```

- `Number.isNaN()` [ES6]
  - does not coerce

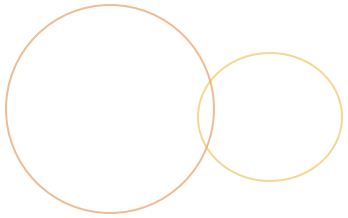
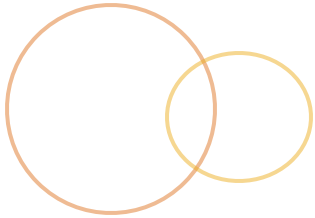
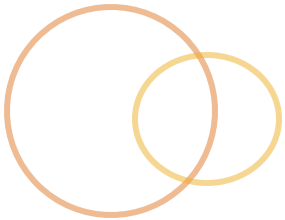
Falsy / Truthy

- Really just *coercion*
- These coerce to **false**
  - `false`
  - `null`
  - `undefined`
  - `""`
  - `0`
  - `NaN`
- Everything else coerces to **true**, including…
  - `{}`
  - `[]`
    - `[] == true // false`
    - `[] == false; // true`
  - "0"
  - "false"

Exercise - Truthing and Falsing

- Two things to ponder:
  - Is the expression **truthy or falsy**
  - and what is the **actual result** of the expression

```
1.null                          1.falsy
2.true                          2.truthy
3.true && 5 && 10               3.truthy
4.1 && false && 2               4.falsy
5.false || 2                    5.truthy
6.x = 2                         6.truthy
7.10 >= 5                       7.truthy
8.1 || 2 || 3                   8.truthy
9.[]                            9.truthy
```


module

SCOPE

Scope

- Variable **access** and **visibility** in a piece of code at a given time
- **Scope is Lexical** (static)
  - as opposed to *dynamic*
  - Scope is defined at author-time
  - No need to execute; you can read code and determine scope
- Three scopes to consider in JavaScript
  - Function Scope
  - Global Scope
  - Block Scope [ES6+]

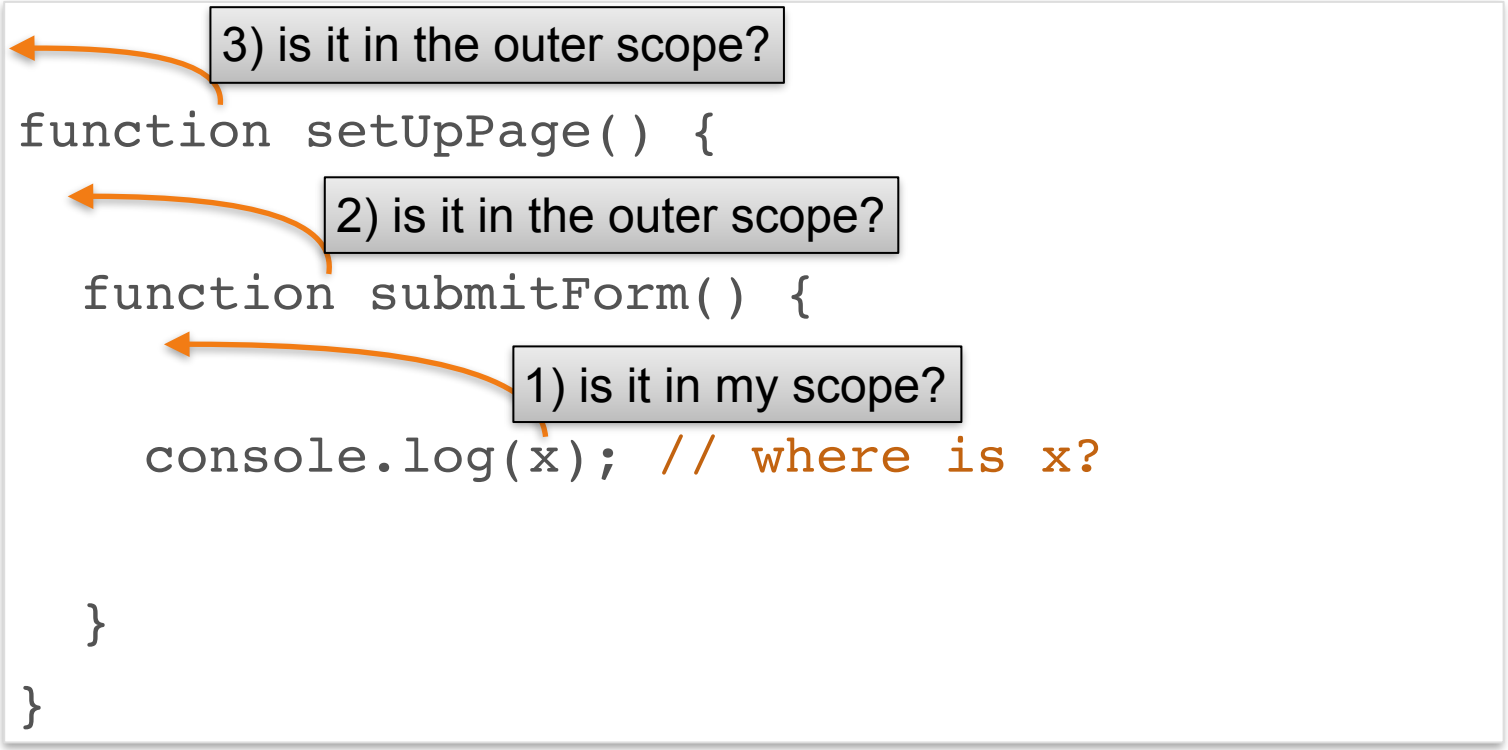
Function Scope

- JavaScript is originally **function-scoped**
  - **var** declares a variable in current function scope
  - variable is said to be “local” to the function

```
var x = 10; // what is the scope?

if (x > 1) {
  var y = 12; // scope?
}

function doMath(x) {
  var y = 10; // scope of x and y?
}
doMath(5);
```

Scope chain

- When a variable is not found in the current scope…
  - JavaScript will look into the outer scope
  - All the way up the scope chain until global



Scope visibility

- Outer scopes **can not** access inner scopes

```
function doSomething() {
  var y = 10;
}

// can I access y?
```

- Inner scopes **can** access outer scopes

```
var x = 10;
function doSomething() {
  // can I access x?
}
```

What scope?

- What are the scopes here?

```
var a = 5;
function foo(b) {
  var c = 10;
  d = 15; // where is d?

  function bar(e) {
    var c = 2; // which c?
    a = 12; // which a?
  }
}
```

What scope, pt 2?

- What are the scopes here?

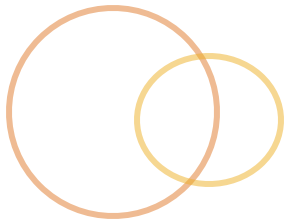
```
var a = 5;
function foo(b) {
  var c = 10;
  d = 15; // where is d?

  if (d < 5) {
    var c = 2; // which c?
  }
}
```

Block scope in ES6+

- **`let` & `const`** define variables in block scope
  - same visibility rules apply
  - can't redeclare (this is nice)
  - `let` is mutable while `const` is immutable*
  - won't allow auto-definition in the global object
  - *in practice: use `const`, unless you need to mutate*

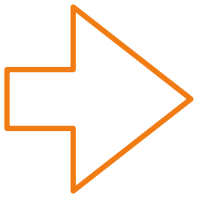
```
let x = 5;
if (x > 1) {
  let x = 10; // This is OK, shadows outer x
  let y = 20;
}
console.log(x); // 5
console.log(y); // ReferenceError - not defined
console.log(window.x); // undefined
```

The Global Scope



- Refers to the outermost object
  - In a browser, this is `window`
- Variables are set in global when
  - Declared w/out “var”
  - Declared outside of any function or block
- Don’t muddy up your global scope
  - ‘use strict’;
  - let or const

```
x = 12;
var y = 1;
function setter () {
  z = 100;
}
```



```
const x = 12;
const y = 12;
function setter () {
  var z = 100; // or const
}
```

Strict mode

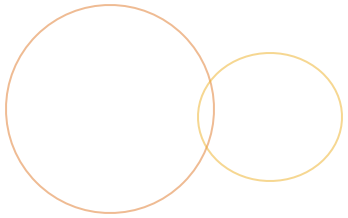
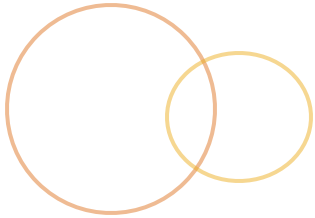
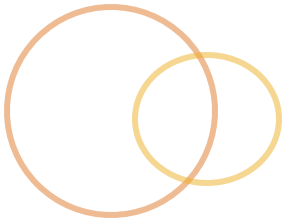
- Opt in to a more **restrictive ES5**
  - It kills deprecated and unsafe features
  - It changes "silent errors" into thrown exceptions
  - Prevents global scope auto-setting
- Can be set **globally** or within **function** block
  - Careful when concatenating scripts

```
// entire script
'use strict';


// or just per function
function whatever() {
  'use strict';
}
```

Exercise – Function Scope

- **Scope Sharing**
  Write two functions that share a *global* and *non-global variable*
  - Fork me
    - http://jsfiddle.net/mrmorris/nv348zo4/

Solutions:
Scope Sharing: http://jsfiddle.net/mrmorris/ksy6js0e/


module

HOISTING

Exercise: Hoisting (pt 1 of 3)

- What will the output be?

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

 console.log(x); // what will the output be?
 return x;
}

foo();
```

Exercise: Hoisting (pt 1 of 3)

This…

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

 console.log(x);
 return x;
}
foo();
```

Becomes…

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

 console.log(x); // 42
 return x;
}
foo();
```

Exercise: Hoisting (pt 2 of 3)

- And this?

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

Exercise: Hoisting (pt 2 of 3)

This…

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

Becomes…

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

Exercise: Hoisting (pt 3 of 3)

- And finally

```
foo(); // ?
bar(); // ?


function foo() {
 console.log("Foo!");
}


var bar = function(){
 console.log("Bar!");
}
```

Exercise: Hoisting (pt 3 of 3)

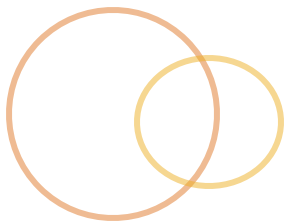
This…

```
foo();
bar();


function foo() {
 console.log("Foo!");
}


var bar = function(){
 console.log("Bar!");
}
```

Becomes…

```
var bar;
function foo() {
 console.log("Foo!");
}

foo(); // Foo!
bar(); // TypeError


bar = function(){
 console.log("Bar!");
}
```

Hoisting



- When a variable declaration is **lifted** to the top of its scope
  - … only the declaration, not the assignment
  - JS breaks a variable declaration into two statements
- **Best practice**
  - declare variables at the top of your scope

This…

```
var myVar = 0;
var myOtherVar;
```

Is interpreted by JS as…

```
var myVar = undefined
var myOtherVar = undefined;
myVar = 0;
```

Function hoisting

- Function *statements* are hoisted, too

```
hoo(); // 'hoo'
bat();  // TypeError, function not defined

function hoo() {
   console.log('hoo');
}

var bat = function() {
   console.log('bat');
}
```

Hoisting with `let` & `const`

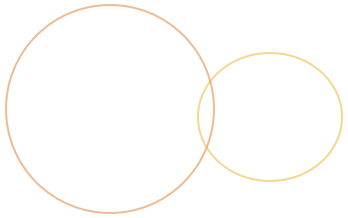
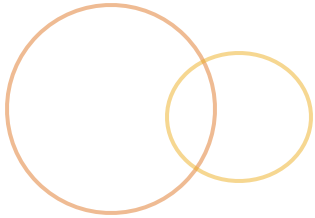
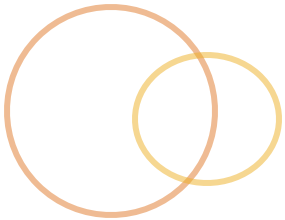
- Variables declarations with **`let`** and **`const`** are not hoisted
  - **Temporal Dead Zone** between declaration and having a value set results in ReferenceErrors
  - const variables ***must*** be declared with a value, however

```
console.log(x); // ReferenceError
let x = 5;

// when using an outer scoped y to set inner
let y = y + 5; // ReferenceError

const z; // SyntaxError
const z = 5; // OK
```

Exercise: Clean that scope

- **Operation cleanup**
  Fix a script that is broken due to mis-used scope and hoisting
  - Fork me:
    - http://jsfiddle.net/mrmorris/s5wLyptf/

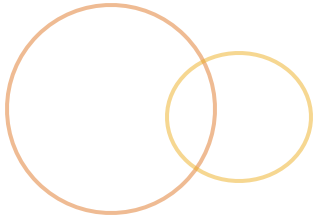
Solutions:
Operation cleanup http://jsfiddle.net/mrmorris/b6qnnk06/

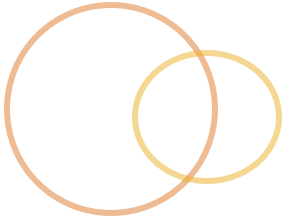





module

OBJECTS

Objects





- Remember that everything is an object except **`null`** and **`undefined`**
  - Even primitive literals (numbers, strings, etc) have object wrappers
- An object is a dynamic collection of properties

```
var dog = {
 name: 'Fido',
 age: 10
}
dog.speak = function() {
  console.log('Bark!');
}
dog.speak(); // Bark!
```

Why Objects

- Objects are structured data
- Objects as…
  - a collection
  - a map
  - a utility library
- Objects to represent things in our world or system (OOP)
  - They have **attributes** (properties)
  - And **behavior** (methods)
  - And can relate to other objects

Four ways to create an object

- Object **literal**

```
var cat = {};
```

We will just be using **literals** today

- A **constructor** function with the `new` keyword

```
var cat = new Cat();
```

- **`Object.create()`**

```
var cat = Object.create(catPrototype);
```

- The **`class`** keyword (and `new`) [ES6+]

```
class Cat {}
var cat = new Cat();
```

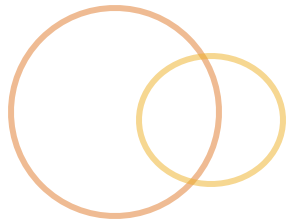
The Object Literal

- Create an object literal with {}:

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

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

Object properties



- Can get/set with dot or array-access syntax

```
myObj.key;
myObj.key = 5;

myObj[“key”];
myObj[“key”] = 5;

var propName = “key”;
myObj[propName] = 5;
```

- Can delete a property with `delete`

```
delete myObj.key;
```

Object reflection

- Objects **inherit** properties from their prototype
  - ex: Array inherits from Object
  - "**Own**" means the property exists on the object itself, not from up the **prototype chain**
  - Use **`in`** and **`hasOwnProperty`** to determine where property resides

```
var myObj = { name: 'Jim' };
myObj.toString(); // [object Object]

'name' in myObj; // true!
'toString' in myObj; // true
myObj.hasOwnProperty('toString'); // false!
```

Object reflection, continued

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

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
  - Their values (properties) can change
  - Pass by reference

Exercise - Mutations

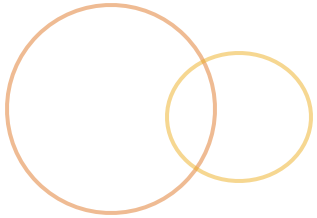
- What will the result of this be:

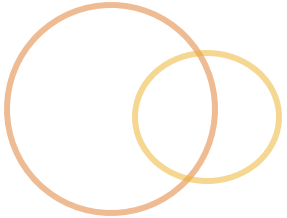
```
var rabbit = {name: 'Tim'};
var hp = 100;

function attack(obj, hp) {
  obj.fight = true;
  hp = 10;
}

attack(rabbit);
console.log(hp, rabbit); // ???
```

Enumerating over objects

- `for…in`
  - Over object properties
- `for…of (ES6)`
  - Over *iterable* values
- ~~`for each…in`~~
  - deprecated
  - over object properties

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)`
  - In order of insertion of the property

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


for (var prop in obj) {
  if (obj.hasOwnProperty(prop)){
    console.log(prop);
  }
  obj[prop];    // true
} // outputs: foo, bar
```

for...of [ES6]

- Loop over ***enumerable values*** of an **iterable**
  - Will include inherited properties as well, including stuff you probably don't want
  - **Not just objects** — *iterables* (including arrays)

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

for (let val of iterableThing) {
  console.log(val);
} // true, false


for (let x of [1,2,3]) {
  console.log(x);
} // 1, 2, 3
```

Properties descriptors

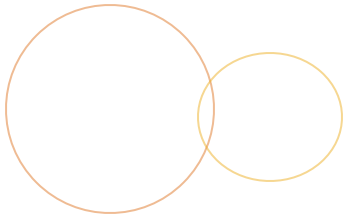
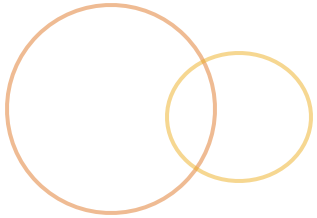
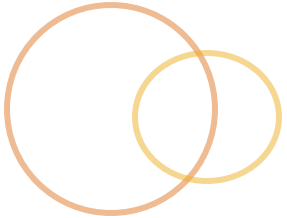
- Object properties have **descriptors**
- They modify property behavior

```
var myObj = {};
Object.defineProperty(myObj, “key”, {
    value: 5,
    enumerable: true, // included in loop
    configurable: false, // re-configurable
    writable: false, // re-assignable
    // get: function() {return ‘hi’;}
}
myObj.key = 10; // silently fails
```

Exercise: Copy Object

- **Object Copy**
  Create a function that can clone an object's own properties
  - Fork me:
    - http://jsfiddle.net/mrmorris/mLccst8c/

Solutions:
Copy Object http://jsfiddle.net/mrmorris/qwcLhf69/

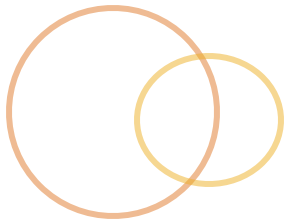

module

FUNCTION BASICS

Functions: "The best part of JS"

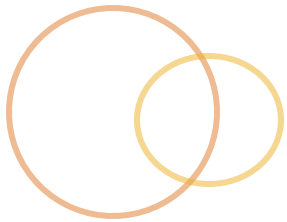
- Reusable, callable blocks of code
- Functions can be used as:
  - Object methods
  - Object constructors
  - Modules and namespaces
- They *are* ***First Class Objects***
  - *Can have their own properties and methods*
  - *Can be passed as function arguments (higher order!)*
  - *Can be referenced by variables*

Defining a function



- Four ways
  - Function **declaration**
  - Function **expression**
  - `Function()` **constructor**
  - **Fat arrow** [ES6+]
- A bunch of examples:
  - http://jsfiddle.net/mrmorris/N8vcg/

Function Declaration



```
// declaration
function adder(a, b) {
      return a + b;
}


// invokation
adder(1, 2); // 3
```

- The function name is *mandatory*
- Function declarations are ***hoisted*** to the top of the scope; available for entire scope

Function Expressions

```
// function expression
var adder = function(a, b) {
      return a + b;
}

// invokation is identical
adder(1, 2); // 3
```

- Define a function and assigns it to a variable
- Function name is optional — *making it anonymous*

Anonymous and named

```
// anonymous function
var funcRef = function() {};

// named anonymous function
var recursiveFunc = function me(a) {
  // *name is scoped to inner function
  me(a++);
}
```

Anonymous functions

- 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)
  - Aren't hoisted

Invokation

- *Run* (invoke) the function with `()`
  - `myFunctionName(argument1, argument2);`
- Missing arguments are set as `undefined`

Default Values [ES6]

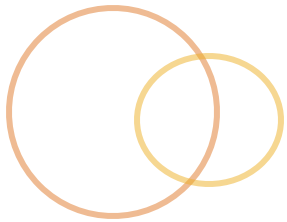
- ES6

```
function adder(first, second = 1) {
  // body
}
function addComment(comment = getComment()) {
  // body
}
```

- Pre-ES6

```
function adder(first, second) {
  second = second || 1;
}
```

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;
```

Function `arguments`

- Functions have access to a special internal when invoked, **`arguments`**
  - contains all parameters passed to the function
  - an *array-like* object
    - needs to be converted to an array to get all the array-methods

Function **arguments**

```
function sumAll() {
  // call an array method with
  // with arguments as the function context
  var args = Array.prototype.slice.call(arguments);

  // or in ES6
  var args = Array.from(arguments);

  return args.reduce(function(acc, curr) {
     return acc + curr;
  });
}
sumAll(1, 2, 3); // ?
```

Functions as First Class Objects

```
// function passed in to another function
setTimeout(function() {
  console.log('HI!');
}, 1000);


// check the docs; we define argument names
[1,2,3].forEach(function(curr, i, arr) {
  console.log(curr, i, arr);
});
```

- Functions can be passed around as arguments
- We can define argument names when we define per an api/interface

(Lots of) global functions

- **alert(msg);**
- **confirm(msg)**
- **prompt(msg, msg);**
- isFinite()
- ~~isNaN()~~ // use Number.isNaN() [ES6]
- **parseInt()**
- parseFloat()
- encodeURI(), decodeURI()
- **setInterval, clearInterval**
- **setTimeout, clearTimeout**
- eval(); // dangerous

Timer functions

- Establish **delay** for function invokation

```
// invoke func in 500 milliseconds
var timer = setTimeout(func, 500);
clearTimeout(timer); // cancel
```

- Establish an **interval** for periodic invokation

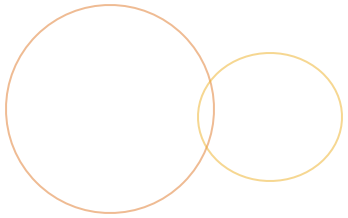
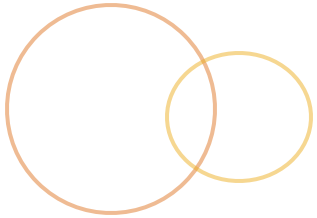
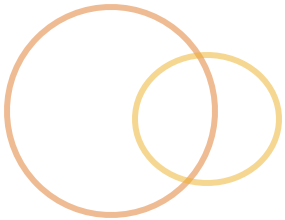
```
// invoke func every 1 second
var timer = setInterval(func, 1000)
clearInterval(timer); // cancel it
```

- Context will always be global for the callbacks
- http://jsfiddle.net/mrmorris/s5g2moc6/

Exercise – Functional FizzBuzz

- **FizzBuzz Function**
  Create a function that:
  - Accepts a single number argument and logs the proper FizzBuzz result for that number
  - Loop through numbers 1-100
  - Consider: How would you test this?
  - **The rules of FizzBuzz**
    - 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/mrmorris/raosjdmq/

Solutions:
FizzBuzz Function http://jsfiddle.net/mrmorris/3gxvkmou/

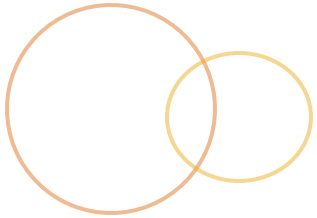

module

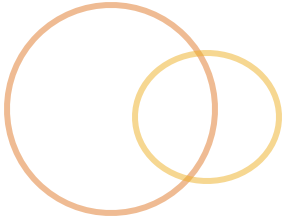
CONTEXT

Scope & Context

- We already discussed ***Scope***
  - Determines visibility of variables
  - Lexical scope (write-time)
- There is also ***Context***
  - *R*efers to the location a function/method was invoked *from*
  - Like a *dynamic scope*; it is defined at run-time
  - Context is referenced by a keyword in all functions: `this`

this?





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

```
function runMe() {
   console.log(this);
}

runMe(); // ?
```

`this` is context

- Reference to an object
  - The ***context*** where the function is running
  - *"The object of my invokation"* 🌹
- Dynamically bound
  - Determined on invokation
  - Not lexical
- Basis of
  - Inheritance
  - Multi-purpose functions
  - Method awareness of their objects

`this` example

```
var person = {
  name: “Carol Danvers”,
  speak: function() {
    console.log(“Hi, I am”, this.name);
  }
}


person.speak(); // ?
var speak = person.speak;
speak(); // ?


// and if we put it on another object?
var otherPerson = {name: “Jim”}
otherPerson.speak = person.speak;
otherPerson.speak(); // ?
```

Binding context

- Default binding
  - Global
- Implicit binding
  - Object method
    - Warning: Inside an inner function of an object method it refers to the global object
- Explicit binding
  - Set with `.call()` or `.apply()`
- Hard binding
  - Set with `.bind()`
- Constructorbinding with “`new`” keyword
- http://jsfiddle.net/mrmorris/RUNS5/

“this” and global

- It’s possible to “leak” and access the global object when invoking functions that reference this from outside objects

```
var setName = function(name) {
  this.name = name;
}
setName('Tim');
name; // "Tim"
window.name === name; // true! oops.
```

- “use strict” prevents leaks like that by keeping global “this” undefined in this case

Explicit binding

- Context can be changed via a Function's `call`, `apply` and `bind` methods

```
obj.foo(); // obj context
obj.foo.call(window); // window context
```

- "`bind`" returns a copy of the function with the context re-defined.

```
var getX = module.getX;
boundGetX = getX.bind(module);
```

- http://jsfiddle.net/mrmorris/or7y5orn/

Example: Explicit binding

```
var speak = person.speak;

// invoke speak in the context of person
speak.call(person);
speak.apply(person);

// invoke speak in the context of otherPerson
person.speak.call(otherPerson);
```

Example: Binding context

```
// permanently bound to person object
var speak = person.speak.bind(person);
speak();

// and if we put it on another object?
var otherPerson = {name: "Jim"};

otherPerson.jimSpeak = person.speak.bind(person);
otherPerson.jimSpeak(); // ?
```

Arrow Functions [ES6]

- **(Fat) Arrow** functions
  - Super short function syntax
  - Always anonymous
  - Lexical contextual binding
- Caveats
  - No **`arguments`** of its own (the *outer* function's args)
  - No **`this`** of its own (uses the enclosing context)

```
var add = function (x) {
  return x + 1;
}

// can instead be written as
var add = x => x + 1;
```

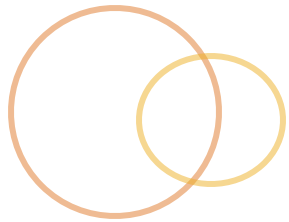
Arrow functions continued

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

// becomes
var add = (x, y) => x + y;

// which is also
var add = (x, y) => {
  return x + y; // what is this here?
}

me = {
  name: 'Tim',
  talk: (x) => {
    console.log(this.name, x); // this is global :(
  },
  talkLater: function () {
    setTimeout(() => {console.log(this.name)}, 1000); // this is me :D
  }
}
```

"The same `this` inside the function as outside the function".

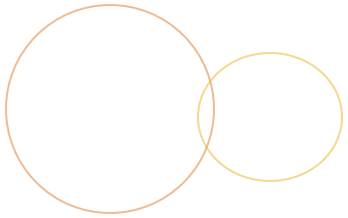
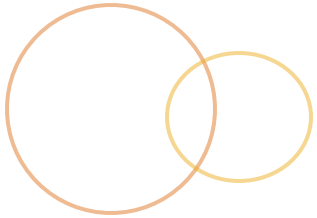
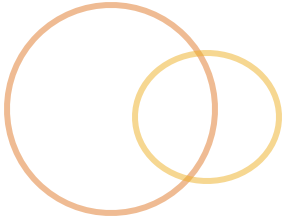
Bound on creation (not invokation)

Exercise - Objects



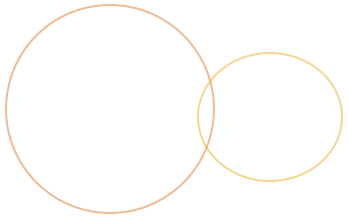
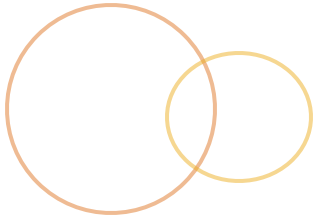
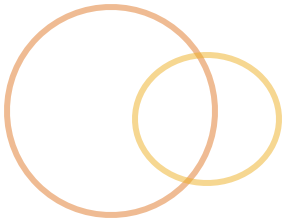
- **Objectify Yourself**
  - Fork: https://jsfiddle.net/mrmorris/rt5z9mo0/

Solutions:
Objectify Yourself - https://jsfiddle.net/mrmorris/d2847z01/


break - did we get this far?

INTENDED END OF DAY 1


module

IFFE

IIFEs

- **I**mmediately **I**nvoked **F**unction **E**xpression
- A function that is defined within a parenthesis, and immediately executed

```
(function() {
  var x = 1;
  return x;
})();
```

IIFE Uses

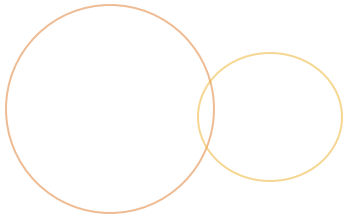
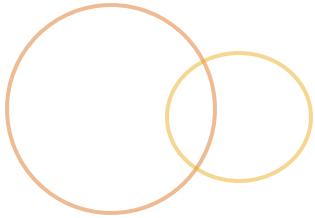
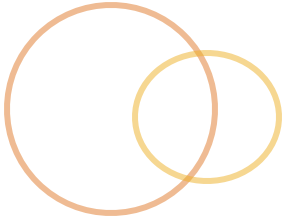
- Define namespaces/modules/packages
  - Typically singletons or “static” objects
- Creates a scope for private variables/functions
- Extremely common in JS

Privacy and modules with IIFEs

```
var helper = (function() {
  var x = 1; // effectively private
  return {
    getX: function() {
      return x;
    },
    increment: function() {
      return x = x + 1;
    }
  }
})();

helper.getX();
helper.increment();
```

Privacy and modules with IIFEs

```
var helper = (function($) {
  var $el = $('button');
  return {
    getElement: function() {
      return $el;
    },
    clearElement: function() {
      $el.html('');
    }
  }
})(jQuery); // pass in globals
```


module

CLOSURES

Closures

- A **closure** is created when an inner function has access to an outer (enclosing) function's variables
- A function that maintains state (it’s outer scope) after returning
- It has access three scopes:
  - Own – variables defined in its body
  - Outer – parameters and variables in the outer function
  - Global
- Pragmatically, *every* function in JavaScript is a closure!

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

Consider scope:

```
var a = 1; // global

function accessA() {
     console.log(a); // ok
};

a = 5;
accessA(); // 5 !
```

Close over scope

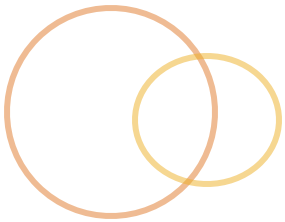
```
function closingOver() {
  var a = 1; // local

  return function accessA() {
      console.log(a);
  };
}
accessA = closingOver();
a = 5;
accessA(); // ?
```

http://jsfiddle.net/mrmorris/974U2/

Closure Module Example

```
var helper = (function() {
  var secret = “I am special”;

  return {
    secret: secret,
    tellYourSecret: function() {
      console.log(secret);
    }
  }
})();

helper.tellYourSecret(); // ?
helper.secret = “New secret”;
helper.tellYourSecret(); // ?
```

Closures for Privacy



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

     var getter = function() {
          return privateVar;
     }


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

Exercise: Closures

- **Month Names**

  Using a closure to track month names in a function

  - http://jsfiddle.net/mrmorris/y37qch2g/

- **Objectify Me - Private Trophies**

  - In your Objectify Me lab, go back and make “trophies” a private variable with a getTrophy(i) accessor.

- **Counter Object**

  - Make a function that stores a “count” which can be increased or decreased.
    https://jsfiddle.net/mrmorris/yn7ywv7q/

Solutions:
Month Names - http://jsfiddle.net/mrmorris/507kocdn/
Objectify Yourself private - https://jsfiddle.net/mrmorris/wocw3b1v/
Counter Object - https://jsfiddle.net/mrmorris/8r9n4yp1/

Function Chaining

- Fluent style of writing a series of function calls on the same object
  - By returning context (**`this`**)

```
"this_is_a_long_string"
    .substr(8)
    .replace('_', ' ')
    .toUpperCase(); // A LONG STRING
```

Support function chaining

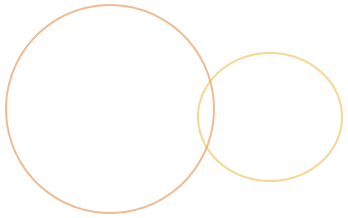
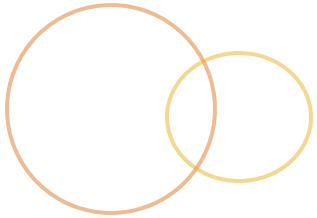
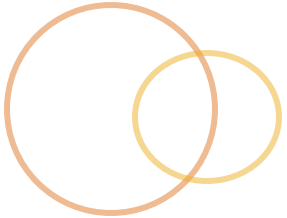
```
var Cat = {
      color: null,
      hair: null,
      setColor: function(color) {
            this.color = color;
            return this;
      },
      setHair: function(hair) {
            this.hair = hair;
            return this;
      }
};

Cat.setColor('grey').setHair('short');
```

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();
  }
  ```
  ```

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/
      ```

Functions Recap

- Are **Objects** with their own methods and properties
- Can be **anonymous**
- Can be bound to a particular **context**, or particular **arguments**
- Can be **chained** together, provided the return of each function has methods
- **Closures** can be used to maintain access to calling context's variables
- **IIFEs** can be used to maintain internal state
  - Both closures and IIFEs can be used to simulate "private" or hidden variables


the end is near

WRAPPING UP

Final Exercise

- **Track Temperatures**
  Build a mini module to store and track temp values
  - https://jsfiddle.net/mrmorris/3s0sgk9e/
- **Smart Stub**
  - Write a function that keeps track of how many times it has been called, as well as the arguments it was called with in sequence
  - https://jsfiddle.net/mrmorris/zyqd0cou/

Solutions:
Track Temps - https://jsfiddle.net/mrmorris/uz2bh5jr/
Smart Stub - https://jsfiddle.net/mrmorris/jeevsryx/

Going beyond

- Inheritance (Prototype)
- Advanced Modules
- Promises and asynchronous JS
- AJAX
- Observables
- JS in the Browser
  - The DOM
  - Events
- JS in the server
  - NodeJS

Best Practices so far…

- “use strict”
- Don’t pollute global
- Take care of scope; define variables up top
- Determine a nice code standard and stick to it
- Use semi-colons (or… don’t)
- Take care with coercion; use strict comparison
- Avoid primitive constructors (ex: Number() and String())
- Use ES6 standards if you’re able… or babel to transpile
  - let/const
  - fat arrow only when it’s useful

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

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**