

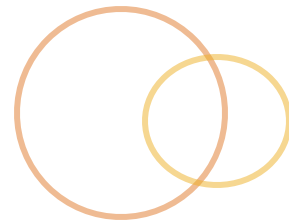
JS Fundamentals

Core JavaScript

Ryan Morris
@mrmorris



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?

Class Outline



Day 1

- 🕒 Introductions/Setup
- 🕒 JavaScript Syntax & Data types
- 🕒 Hoisting, Scope and Context
- 🕒 Prototype / OO

Mix of ES5/6
No OO
No modules

Day 2

- 🕒 Refresher of HTML/CSS
- 🕒 The DOM
- 🕒 Event handling
- 🕒 Ajax & Promises

Any specific topics you're interested in?

Expectations



- ◎ We're going to cover a **lot of ground**, quickly
 - ◎ *Don't expect to remember everything*
- ◎ I am a **guide**, not a deity
 - ◎ *I learn & re-learn things every day*
- ◎ Become an excellent **researcher**
 - ◎ *The documentation and google are our friends*
- ◎ There is **rarely "one way"** to do a thing
 - ◎ *Get used to opposing opinions and ambiguity*

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**
- ◎ Don't be afraid to **break stuff**

Resources



🕒 Documentation

- 🕒 <http://devdocs.io>

- 🕒 <https://developer.mozilla.org/en-US/docs/Web>

- 🕒 Google it.

🕒 Compatibility checks

- 🕒 <http://caniuse.com>

🕒 ES 5 compatibility table

- 🕒 <http://kangax.github.io/compat-table/es5/>

🕒 Read up

- 🕒 <https://github.com/leonardomso/33-js-concepts>

- 🕒 You Don't Know JS

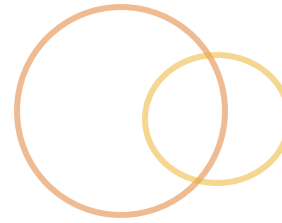
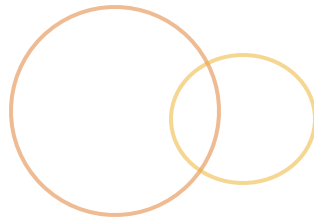
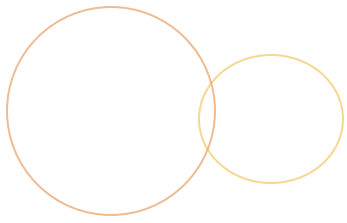
- 🕒 JavaScript: The Good Parts

Set up for our labs

Don't have **NodeJS**?
Head here to download:
<https://nodejs.org/>

- 🕒 *In your command line...*
- 🕒 **Download** or clone this repository
 - 🕒 <https://github.com/rm-training/web-dev-bc>
- 🕒 **Initialize** the project
 - 🕒 `npm install`
- 🕒 **Start** the server
 - 🕒 `npm start`
 - 🕒 Visit <http://localhost:3000/>
- 🕒 Let's **explore** our project files
 - 🕒 Set up a new workspace in your IDE

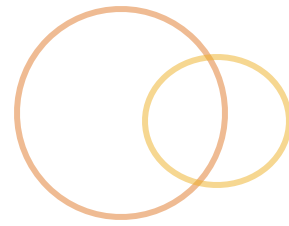
everyone all set with the above?



module

JAVASCRIPT INTRO

What is JavaScript?

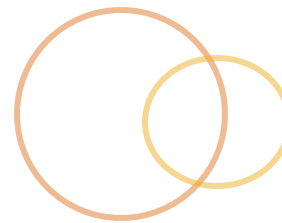


- ◎ **Interpreted**
- ◎ Case-sensitive C-style syntax
- ◎ Dynamically typed (with weak typing)
- ◎ Fully **dynamic**
- ◎ **Single-threaded** event loop
- ◎ **Prototype**-based (vs. class-based)
- ◎ Safe (no CPU or memory access)
- ◎ Depends on the engine + environment running it
- ◎ *Kind of weird but enjoyable*



- ◎ "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 was licensed from *Sun*
 - ◎ Standardized as **ECMAScript**
- ◎ Combines influences from:
 - ◎ Java, "Because people like it"
 - ◎ SmallTalk, prototypal

JavaScript Versions



🕒 ES3/1.5

🕒 Released in 1999 – in all browsers by 2011

🕒 IE6-8

🕒 ES5/1.8

🕒 Released in 2009

🕒 IE9+

🕒 ES6 [ECMAScript 2015] widely supported

🕒 ES7 [ECMAScript 2016] finalized, but weak support

🕒 ES8 [ECMAScript 2017] finalized in June 2017

🕒 ES9 — 2018

🕒 ES.Next...



This is our
sweet spot

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



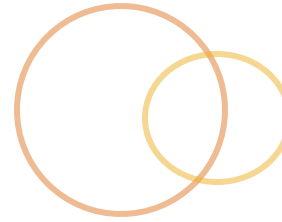
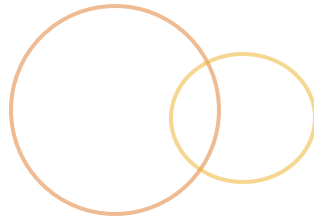
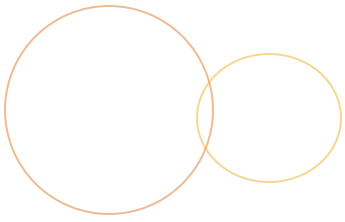
- ◎ It's **not** Java or Class-based
- ◎ Very **dynamic & flexible**
- ◎ Supports **many paradigms**
 - ◎ imperative, functional and object-oriented

Approaching JavaScript



◎ 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



obligatory

HELLO WORLD

Alert hello



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

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

🕒 Alternatively...

- 🕒 In a `<script>` tag
- 🕒 or... in a file linked from an HTML page
- 🕒 or... run by NodeJS

Debugging in the Browser

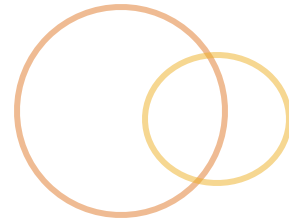


🕒 Browser's JavaScript Console

- 🕒 REPL to experiment and log output
- 🕒 Set breakpoints and monitor variables
- 🕒 Monitor events
- 🕒 View network requests
- 🕒 View memory usage
- 🕒 And of course inspect our HTML/CSS

Let's check it out

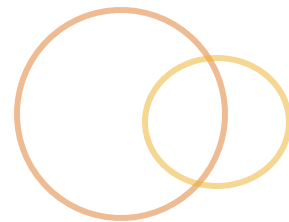
Log hello



Now try

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

The console object



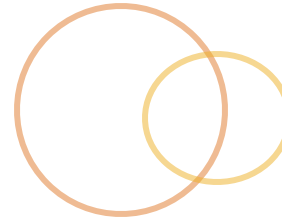
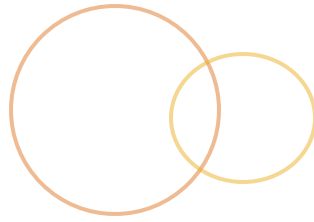
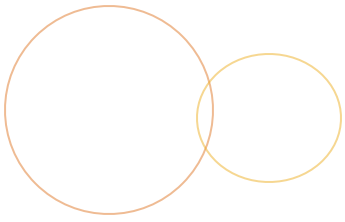
🕒 Console api

```
console.log(msg);      // echo/print/output  
console.assert(data); // for testing  
console.table(data);  // output as table  
debugger;              // triggers a breakpoint
```

🕒 Tips while debugging through the console:

- 🕒 **Clear** your console of old errors
- 🕒 **Check** the error message line reference
- 🕒 **Disable caching**

Everyone do this?



module

SYNTAX BASICS

C-family syntax



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

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

- Spaces, tabs and newlines are **whitespace**.
 - Whitespace and indentation generally don't matter

C-family syntax



- **Block statements** group related statements
- They are wrapped with **curly braces**

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

```
function test() {  
    var x = 5;  
    x++;  
}
```

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

```
var z = {  
    x: 5,  
    y: 10  
}
```

Not a block statement!

Automatic Semicolon Insertion



☉ Semicolons terminate statements

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

☉ They are *mostly* optional

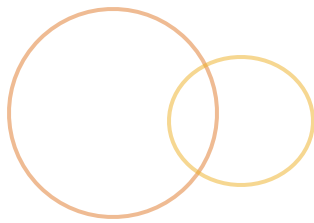
☉ Automatically inserted but not fail-safe

☉ So, don't rely on it...

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

Missing semi-colon here
results in a TypeError

Comments



◎ Lines of text (or code) that are ignored

◎ Many lines

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

◎ Single line

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


Variables



- ⦿ A name to help (or *reference*) a value

```
var MyName = "Ryan";  
var your_name;  
var $; // like jQuery  
var _myName;  
var num10 = 5 + 5;  
var 🍔 = 'burger';
```

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

Declaring variables



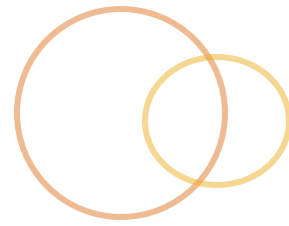
- With the keyword **var**
 - and **let** or **const** in ES6+
- One by one:
- Or in sequence:

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

```
ES6:  
let foo;  
const thing1;
```

```
var a, b;
```

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 = 22; // ok!
```

```
var x; // redeclaring with var has no effect  
x = 10 + x; // x is now 32!
```

Assigning values with let & const



⦿ Cannot redeclare with let

```
let x = 5;  
let x = 10; // error! can't redeclare
```

⦿ Const values are immutable

```
const x; // error! must initialize consts  
const x = 10;  
x = 50; // error! can't modify
```

⦿ However, object properties are still mutable

```
const z = {};  
z.name = "John"; // ok!
```

Default Value



- 🕒 Variables with no value set will default to a special value, undefined

```
var placeholder;  
console.log(placeholder); // undefined  
  
typeof placeholder; // "undefined"
```

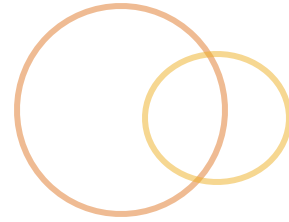
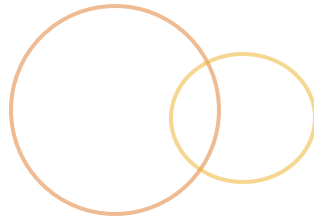
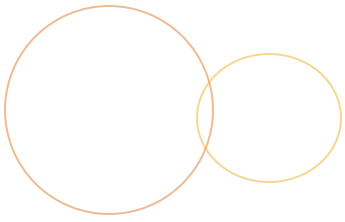
Don't pollute the global scope



- ⦿ Omitting the **var** keyword (or **let**, **const**) creates a ***globally scoped*** variable

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

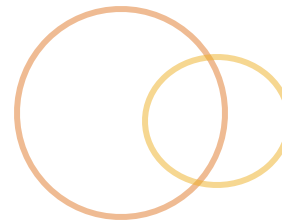
- ⦿ This is bad.



module

DATA TYPES

Primitives



🕒 Five *primitive* data types:

- 🕒 null - *lack of value*
- 🕒 undefined – *no value set* (default)
- 🕒 strings
- 🕒 numbers
- 🕒 booleans
- 🕒 *ES6: Symbol*

🕒 Everything else is an *Object*

- 🕒 ie: Object, Array, Function, Math...
- 🕒 A function is a *callable* object
- 🕒 All *primitives* have *Object* counterparts

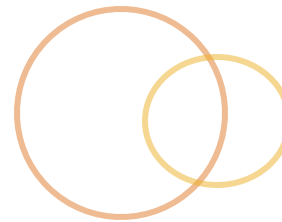
undefined & null



- ⦿ Variables declared without a value will start with `undefined`
- ⦿ Can compare to `undefined` to see if a variable has a value
- ⦿ In practice there is not much difference

```
var a;  
  
// is this variable not yet set?  
a === undefined; // true  
  
// or  
typeof a === "undefined"; // true
```

boolean



🕒 true or false

```
var isRyanTall = true;  
var isSomethingOn = false;  
  
if (isRyanTall) {  
    // do something...  
}
```



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

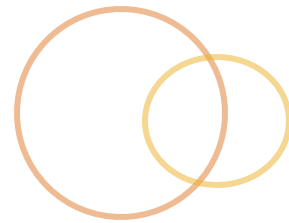
```
var str = "My Name Is";  
var name = 'Ryan';
```

- Combine strings with the + operator

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

- (ES6) Template Literals with ` (*backtick*)

```
var lastName = "Ryan";  
var name = `Hello {x}`; // Hello Ryan
```



number

- 64bit (IEEE 754) floating point
- Numbers can be expressed in many ways:

```
var a = 1; // integer  
var b = 1.5; // decimal  
var c = -3; // negative  
var d = 2.99e8; // scientific  
var e = 0777; // octal  
var f = 0xFF; // hexadecimal, 255  
var g = 0b10000000000000000000000000000000;
```

Number oddities



- There is a max and min value of up to 15 digits
- Decimal arithmetic can be inaccurate

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

- There is a special value for "*not a number*"

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

- And a special value for *infinity*

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

Objects



- ⦿ A list of **key:value** pairs, separated by **commas** and surrounded by **curly braces**

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

- ⦿ Might be considered a *Dictionary*, *Hash* or *Map* in other languages

Objects, continued...



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

Keys are **unordered** strings

Quotes around key names are only required if they include special chars

Values can be **any type of data**, including **functions**

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

Functions



🕒 Functions are **callable objects**

```
function addOne(x) {  
    return x + 1;  
}  
addOne(10); // 11
```

🕒 Functions are for storing some reusable functionality

Functions



- They can be referenced by a **name** or **variable**

```
var hello = function hello() {  
    console.log("Hello!");  
}
```

- They can exist on objects as **methods**

```
var me = {}  
me.hello = function() {  
    console.log("Hello!");  
}
```

- They can take **arguments**

Arrays



☉ Data stored *sequentially*

```
var myThings = []; // empty  
var collection = ['a', 2, 'c', 4];
```

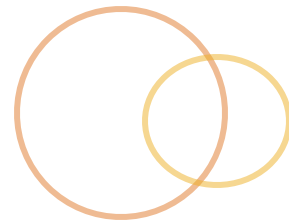
☉ Can access and set values by their *index*

```
collection[1]; // 2  
collection[1] = 20;
```

☉ They are zero-indexed

```
var favColors = ['red', 'blue', 'green'];  
favColors[0]; // 'red'
```

array methods



☉ Arrays are objects...

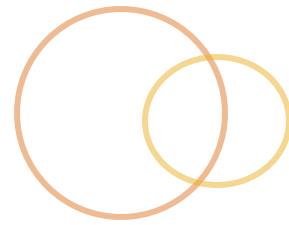
☉ They have additional **properties**

```
myArray.length; // 4
```

☉ And **methods**

```
// adds value to end  
myArray.push( 'John' );  
  
// take value off end  
myArray.pop(); // 'John'
```

Arrays are not arrays



🕒 An **array** is an **object** that behaves *like* an array

🕒 It's a hash-map

🕒 *Not always* stored sequentially in memory

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

🕒 Avoid using strings as keys

🕒 Breaks "length" property

🕒 Avoid heterogeneous array elements

🕒 Performs poorly

Everything* is an object



☉ 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: Super Primitive



- Start the node server

```
$> npm start
```

- Open the following file:

```
public/exercises/primitives/primitives.js
```

- Complete the exercise

- Run the tests by visiting in your browser:

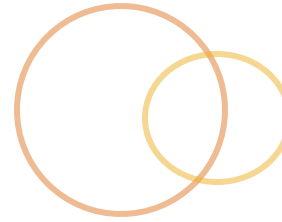
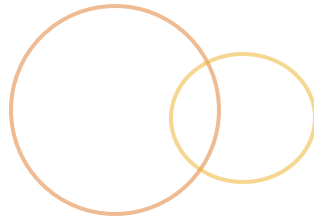
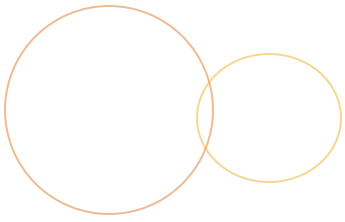
```
http://localhost:3000/exercises/primitives
```

**1) Our test
suite**

2) ESLint

Solution:

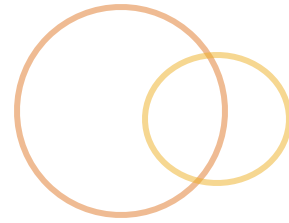
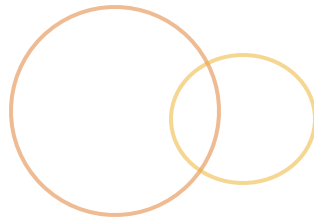
<https://github.com/rm-training/web-dev-bc/blob/master/public/solutions/primitives/primitives.js>



module

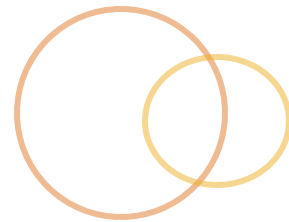
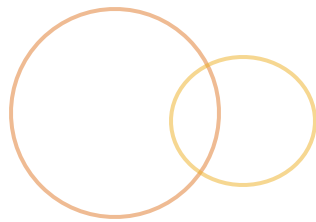
OPERATORS

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 (remainder)

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

Type checking



🕒 **typeof** returns the type of the argument as a string

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

// can be used 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
```

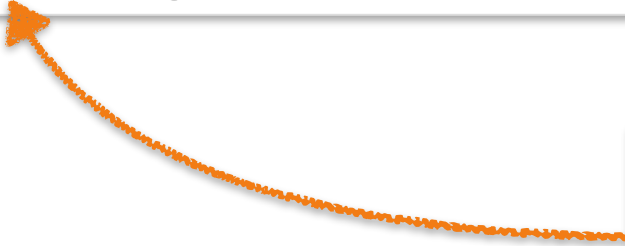
- ◎ *most things, primitives are just **coerced**

Primitive->Object Coercion



- ⦿ This means we can access properties and methods of objects, including primitives

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



JS creates an object wrapper for the primitive, uses it, then throws it away

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"` (or `let`, `const`)
- ⦿ 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: Data Types



- 🕒 Open the following file:

`public/exercises/data-types/index.js`

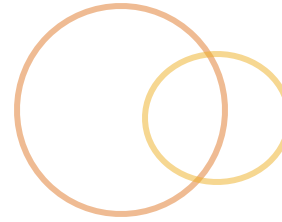
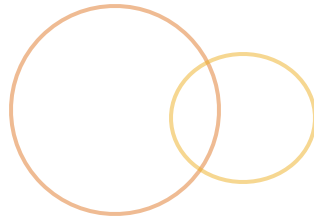
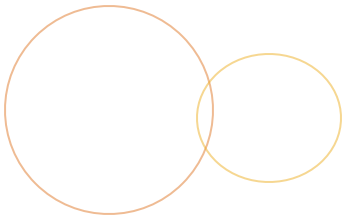
- 🕒 Complete the exercise

- 🕒 View your log output by visiting:

<http://localhost:3000/exercises/data-types>

Solutions:

<https://github.com/rm-training/web-dev-bc/blob/master/public/solutions/data-types/index.js>



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?

Conditional statements



⦿ `if (expression) {...}`

⦿ `if (expression) {`

`...`

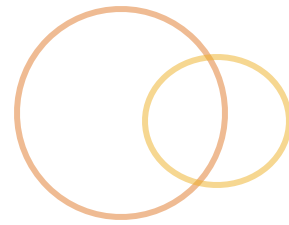
`} else {`

`...`

`}`

⦿ `if {} else if {} else {}`

Relational operators



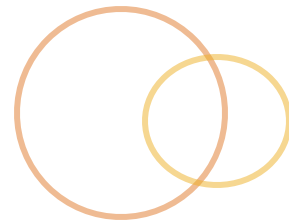
```
'foo' in {foo: 'bar'} // true
[] instanceof Array   // true
5 < 4                 // false
5 > 4                 // true
4 <= 4                // true
5 >= 10               // false
```


Equality operators (strict vs loose)



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

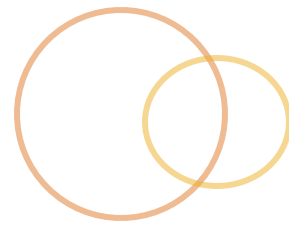
Logical operators



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

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

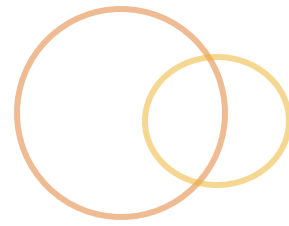
Conditional example



```
// generates a value between 0 and 1
const 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

ES6:

var is scoped to this *block*
let is scoped to each *iteration*

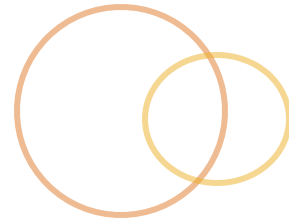
🕒 Do something {x} times

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

🕒 Great for something like... looping over an array

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

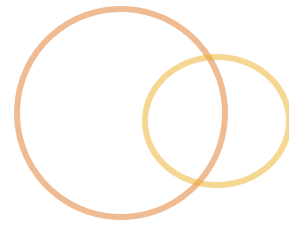
looping - while



```
var i = 0;

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

Break and Continue



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


Exercise: Boolean Operators



🕒 What is the resulting value/output:

```
false && console.log("Yep"); // ?
```

```
true && console.log("Yep"); // ?
```

```
false || console.log("Yep"); // ?
```

```
true || console.log("Yep"); // ?
```

Logical short circuits



⦿ **a && b** returns last true value, otherwise first false

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

⦿ **a || b** returns first true value, otherwise last val

```
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**
- ◎ **Logical short circuits** are a common pattern

Exercise: Control Flow (Level I)



- 🕒 Open the following file:

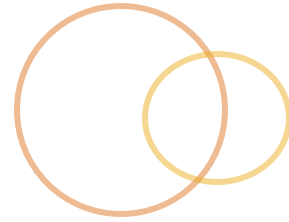
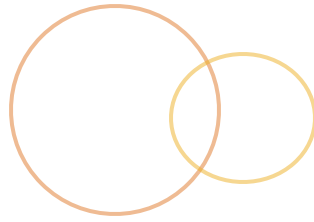
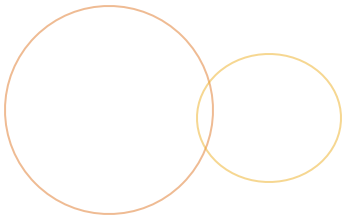
`public/exercises/control/index.js`

- 🕒 Complete the exercise

- 🕒 View test results by visiting:

<http://localhost:3000/exercises/control/>

Solutions: <https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/control>



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"; // "13";
```

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
```

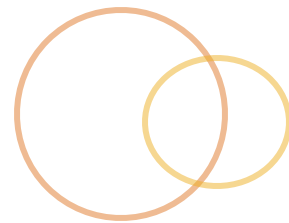

Sometimes coercion is cool



```
// Convert any string to a number  
(+"5"); // 5
```

```
// Convert any value to a boolean  
!![]; // true
```

Coercing to boolean



☉ Most frequently we'll rely on it in logic checks

```
// x is a number
var x = 10;

// if x is true? ... truthy
if (x) {
  // do something
}
```

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 is **true**

```
{ }  
[ ]  
"0"  
"false"
```

Falsy / Truthy



- ☉ Checking falsy-truthy is not always the same as checking equivalency

```
[ ]; // truth  
[] == true; // false  
[] == false; // true
```

- ☉ See:

- ☉ <https://dorey.github.io/JavaScript-Equality-Table/>

Loose vs Strict in Coercion



```
// compare and allow coercion
```

```
5 == "5";
```

```
// compare and don't allow coercion
```

```
5 === "5";
```

Exercise - Truthing and Falsing



Consider **two** things:

Is the expression **truthy or falsy**?

What is the **actual result** of the expression?

1.null

2.true

3.true && 5 && 10

4.1 && false && 2

5.false || 2

6.x = 2

7.10 >= 5

8.1 || 2 || 3

9.[]

Exercise - Truthing and Falsing



Consider **two** things:

Is the expression **truthy or falsy**?

What is the **actual result** of the expression?

1.null

2.true

3.true && 5 && 10

4.1 && false && 2

5.false || 2

6.x = 2

7.10 >= 5

8.1 || 2 || 3

9.[]

1.falsy

2.truthy

3.truthy

4.falsy

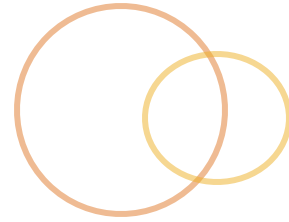
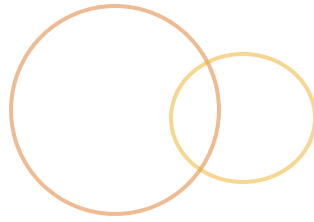
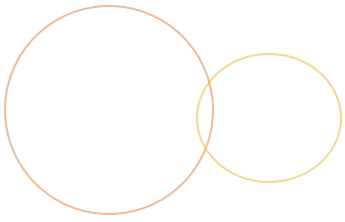
5.truthy

6.truthy

7.truthy

8.truthy

9.truthy



module

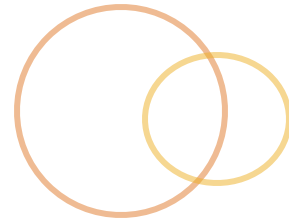
FUNCTION BASICS

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*

Function Declaration



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

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

🕒 The function name is *mandatory*

Function Expressions



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

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

- 🕒 When you assign a function to a variable
- 🕒 Function name is optional — *making it anonymous*

Anonymous can be named



```
// an anonymous function
var someFunction = function() {};

// a named anonymous function
var someFunction = function me(a) {
    // name is available only in inner function
    me(a++);
}
```

Invokation



- Execute a function with `()`
- Pass in any **arguments**
- Missing arguments are set as **undefined**

```
function mult(x, y) {  
  return x * y;  
}
```

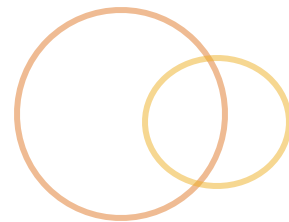
```
mult; // ?
```

```
mult(); // ?
```

```
mult(1) // ?
```

```
mult(1,2); // ?
```

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 value when invoked, **arguments**
- ⦿ Contains all parameters passed to the function
- ⦿ It's an *array-like* object
 - ⦿ Meaning we need to convert it to an array if we want to do "array stuff" on it.

```
function adder(first, second) {  
    // won't work because not an array  
    arguments.forEach(function(el) {  
        console.log(el);  
    });  
}  
  
adder(1, 2); // ?
```

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(current, i, arr) {
  console.log(current, 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()**
- ⦿ **encodeURIComponent(), decodeURI()**
- ⦿ **setInterval, clearInterval**
- ⦿ **setTimeout, clearTimeout**
- ⦿ **eval();** // dangerous

Timer functions



🕒 Establish **delay** for function invocation

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

🕒 Establish an **interval** for periodic invocation

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

Exercise: Functional FizzBuzz



- 🕒 Open the following file:

`public/exercises/fizz-buzz/index.js`

- 🕒 Complete the exercise

- 🕒 **The rules of FizzBuzz**

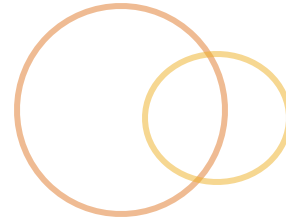
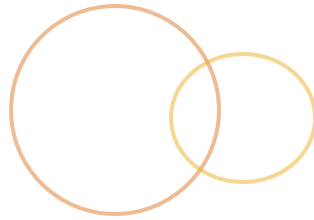
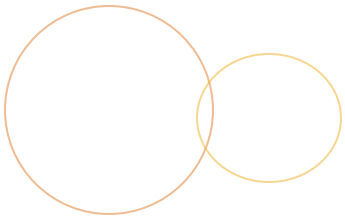
- 🕒 For numbers that are a multiple of 3, log "Fizz"
 - 🕒 For numbers that are a multiple of 5, log "Buzz"
 - 🕒 For numbers that are a multiple of both, log "FizzBuzz"

- 🕒 View your log output by visiting:

<http://localhost:3000/exercises/fizz-buzz/>

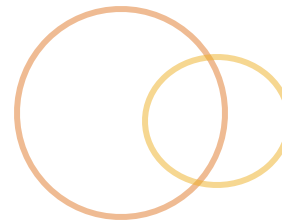
Solutions:

<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/fizz-buzz>



module

SCOPE



- Variable **access** and **visibility**

- Scope is Lexical** (static)

 - as opposed to *dynamic*

 - Scope is defined at author-time

 - You can read code and determine scope w/out running it

- 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?
}

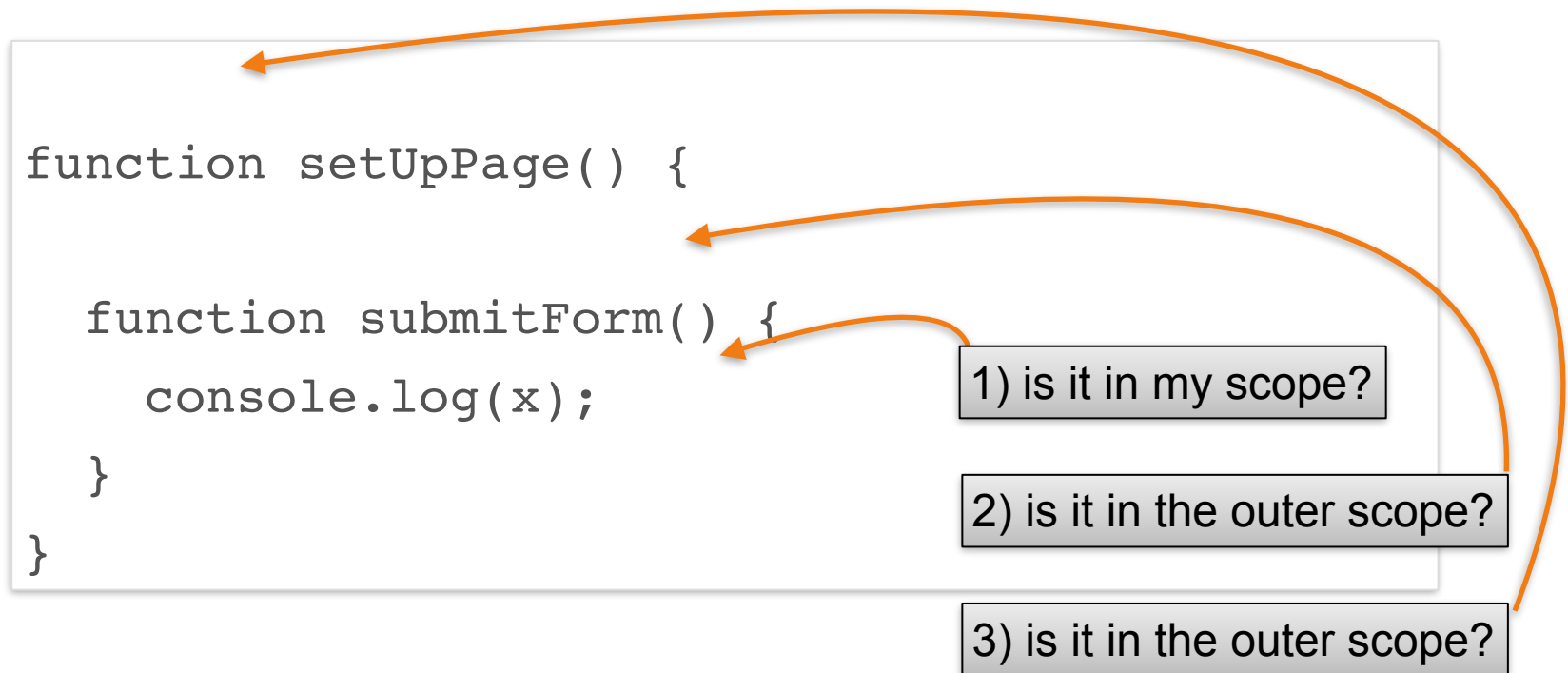
function doMath(x) {
    var y = x + 10; // scope of x and y?
}

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?

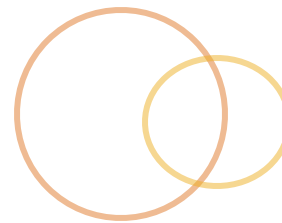


🕒 What are the scopes here?

```
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?

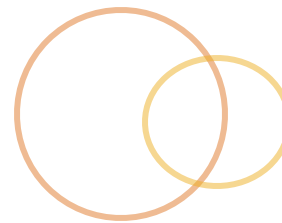


🕒 What are the scopes here?

```
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

Also:

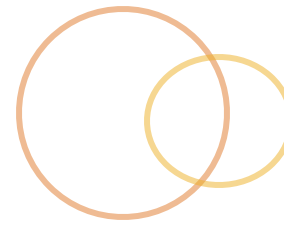
* both can't be redeclare

* **const** is immutable*

* aren't set on global object

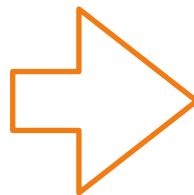
```
let x = 5;
if (x > 1) {
  let x = 10; // This is OK, shadows outer x
  let y = 20;
}
console.log(x); // 5
console.log(y); // ReferenceError - not defined
console.log(window.x); // undefined
```

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 () {  
    const z = 100;  
}
```



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 pollution

☉ 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: Sharing Scope



- 🕒 Open the following file:

`public/exercises/scope/index.js`

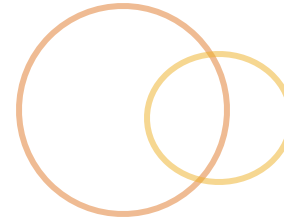
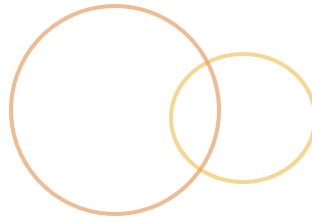
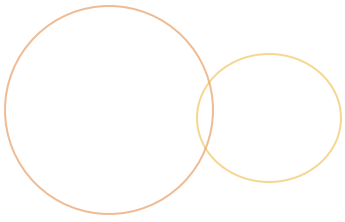
- 🕒 Complete the exercise

- 🕒 Test in the console:

<http://localhost:3000/exercises/scope/>

Solutions:

<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/scope>



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("boy");
}
```

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: Scope Cleanup



- 🕒 Open the following file:

`public/exercises/scope-clean/index.js`

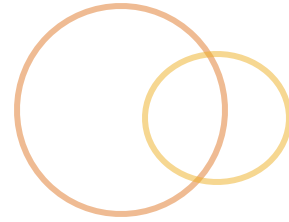
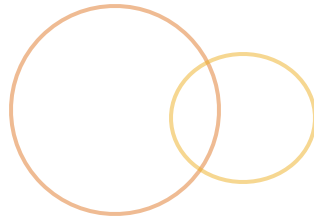
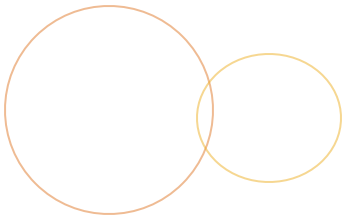
- 🕒 Complete the exercise

- 🕒 View your log output by visiting:

<http://localhost:3000/exercises/scope-clean/>

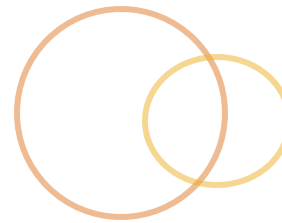
Solutions:

<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/scope-clean>



module

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

```
const cat = {};
```

🕒 A **constructor** function with the new keyword

```
function Animal() {}  
const cat = new Animal();
```

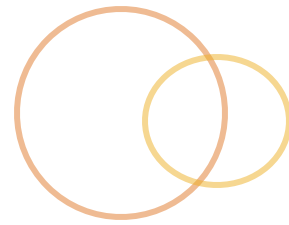
🕒 **Object.create()**

```
const cat = Object.create(Animal);
```

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

```
class Animal {}  
const cat = new Animal();
```


The Object Literal



🕒 Create an object literal with {}:

```
const myObjLiteral = {  
  name: "Mr Object",  
  age: 99,  
  toString: function() {  
    return this.name; // what is this?  
  }  
};
```

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



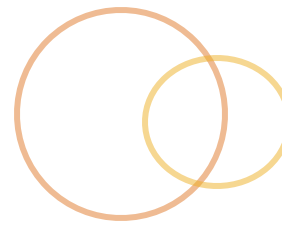
Object.keys(obj)

- Returns array of all "**own**", enumerable properties

Object.getOwnPropertyNames(obj)

- Returns array of all "**own**" property names, including non-enumerable

Mutability - Primitives



- 🕒 All primitives in JavaScript are **immutable**
 - 🕒 Using an assignment operator just creates a new instance of the primitive
 - 🕒 "Pass by value" in functions

```
function addOne(x) {  
  x = x + 10;  
}  
  
let y = 10;  
addOne(y);  
console.log(y); // 10
```

The primitive 10 is passed in as a value only. The original variable "y" is left untouched

Mutability - Objects



🕒 Objects are **mutable**

- 🕒 Their values (properties) can change
- 🕒 "Pass by reference" in functions

```
function addOne(obj) {  
  obj.x = obj.x + 10;  
}
```

The original object is modified by the function.

But.. `obj = {}`
would NOT mutate

```
const obj = {x: 5}; // event as a const!  
addOne(obj);  
console.log(obj); // {x: 15}
```

Exercise - Mutations



🕒 What will the result of this be:

```
const rabbit = {name: 'Tim'};
let attack_count = 0;

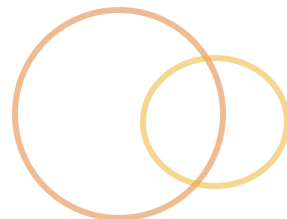
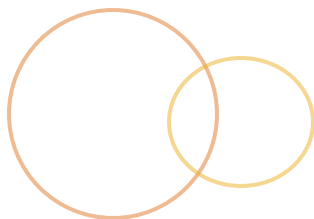
function attack(obj, counter) {
  obj.is_injured = true;
  counter++;
}

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

Enumerating over objects



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



🕒 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

```
const 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)

```
const 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

```
const 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: Copying objects



- 🕒 Open the following file:

`public/exercises/index.js`

- 🕒 Complete the exercise

- 🕒 Run the tests by visiting in your browser:

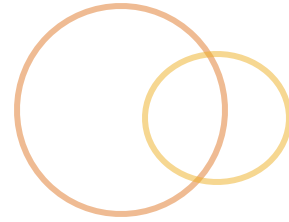
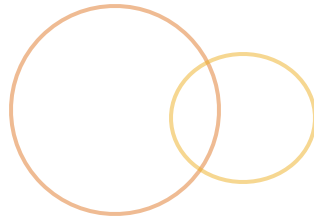
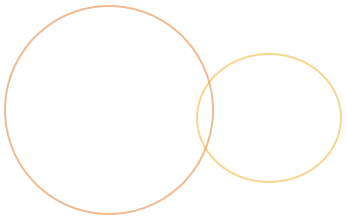
<http://localhost:3000/exercises/copy/>

🕒 Hint: `for (var prop in obj) { /* */ }`

🕒 Hint: `obj.hasOwnProperty(prop)`

Solutions:

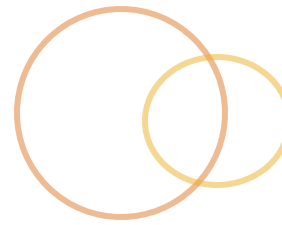
<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/copy>



module

CONTEXT

Scope & Context

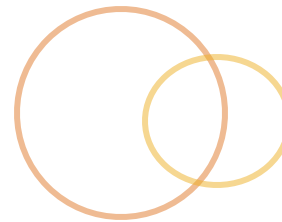
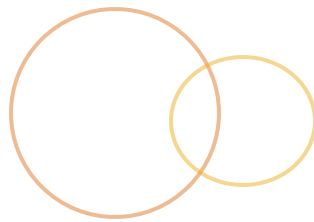


☉ We already discussed **Scope**

- ☉ Determines visibility of variables
- ☉ Lexical scope (write-time)

☉ There is also **Context**

- ☉ Refers 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`

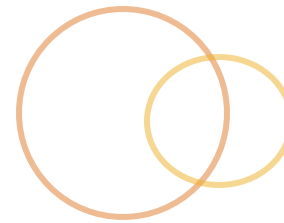


☉ Anyone have an idea what **this** is?

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

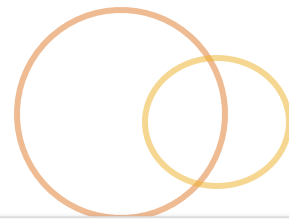
```
runMe( ); // ?
```

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



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

```
person.speak(); // ?
```

```
const speak = person.speak;
```

```
speak(); // ?
```

// and if we put it on another object?

```
const otherPerson = {name: "Jim"}
```

```
otherPerson.speak = person.speak;
```

```
otherPerson.speak(); // ?
```

Binding Context



- ◉ We can control the context that a function is called in
- ◉ **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()`
- ◉ Constructor binding with "new" keyword

"this" and global



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

```
const 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.

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

Example: Explicit binding



```
const 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

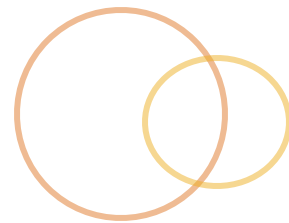


```
// permanently bound to person object  
const speak = person.speak.bind(person);  
speak();
```

```
// and if we put it on another object?  
const 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)

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

// ...can instead be written as...

```
const add = x => x + 1;
```

Arrow function syntax perks



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

// ...written as a fat arrow...

```
const add = (x, y) => x + y;
```

// ...also, written as a fat arrow...

```
const add = (x, y) => {  
  return x + y; // what is this here?  
}
```


Arrow function & a context gotcha

Arrow functions will look to the enclosing, lexical scope of where it was defined to determine which **this** to use.

It is bound on function declaration.

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

Exercise: Objectify Yourself



- 🕒 Open the following file:

`public/exercises/object-you/index.js`

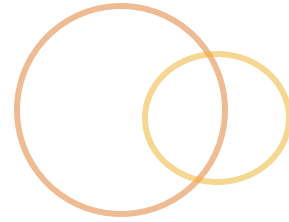
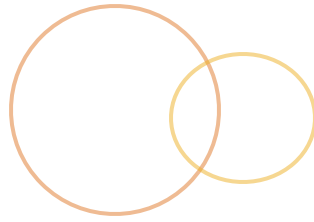
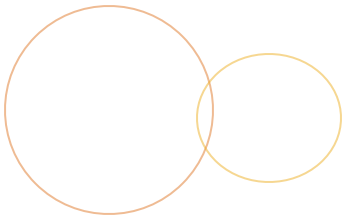
- 🕒 Complete the exercise

- 🕒 View your log output by visiting:

<http://localhost:3000/exercises/object-you/>

Solutions:

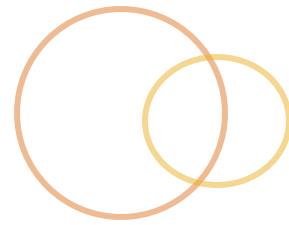
<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/object-you>



module

BUILT-IN OBJECTS

Built-in Objects



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



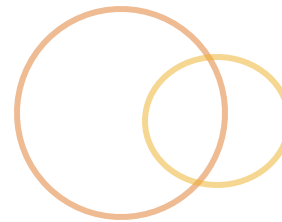
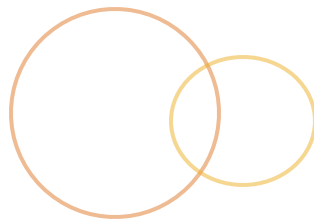
◎ Instance properties

```
"foo".length; // 3
```

◎ Instance method examples

```
let str = "hello";

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



☉ Properties, such as

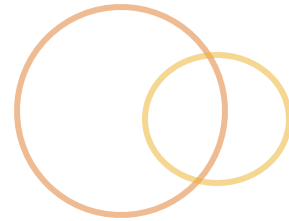
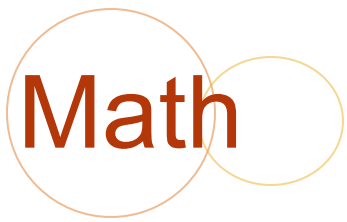
```
Number.MAX_VALUE;  
Number.NaN;
```

☉ Generic methods

```
Number.isInteger()  
Number.isFinite()  
Number.parseFloat()  
Number.parseInt()
```

☉ Instance methods

```
num.toString()  
num.toFixed()  
num.toExponential()
```



◎ Singleton-ish

◎ Methods

◎ `abs, log, max, min, pow, sqrt, sin, floor, ceil, random...`

◎ Properties

◎ `E, LN2, LOG2E, PI, SQRT2...`

Array methods (accessors)



```
let arr = [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 methods (mutation)



```
let arr = [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 iteration methods



```
let arr = [1, 1, 2, 4];
```

```
// where fn is a function
```

```
arr.forEach(fn); // invoke fn for each
```

```
arr.every(fn);   // true if all matches
```

```
arr.some(fn);    // true if any matches
```

```
arr.filter(fn); // new, filtered array
```

```
arr.map(fn);     // new, transformed array
```

```
arr.reduce(fn);  // value from an array
```

Enumerating over an Array



☉ Use "for"

```
for (let i=0; i < myArray.length; i++) {  
    // do something for each element  
}
```

☉ Or .forEach(fn); [ES5+]

```
myArray.forEach(function(val, index, arr) {  
    // do something  
});
```

☉ Don't use "for...in", which doesn't keep keys in order

Array filter, map

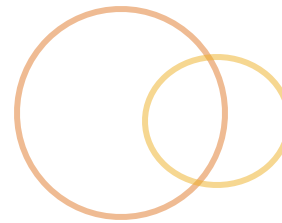


⦿ `Array.prototype.filter()`

- ⦿ Iterate over your array of items passing them to a function. Returning `true` from the function indicates the item should be retained.

```
// ie: remove items that don't equal 2  
myArray.filter(function(item) {  
  return item !== 2;  
});
```

Array map



⦿ Array.prototype.map()

- ⦿ Iterates over array, invoking a function on each value. The return value is the modified value of the item.

```
// ie: increment each value by 1  
myArray.map(function(item) {  
    return item + 1;  
} );
```

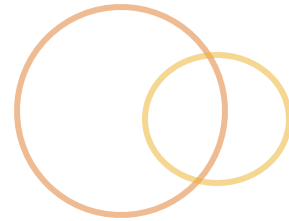
Array reduce()



🕒 `Array.prototype.reduce()`

🕒 Boils down a list of values into a single value.

```
// ie: sum up the array
[0,1,2,3,4].reduce(function(acc, item) {
  return acc + item;
}, 0); // initializer value
```



- 🕒 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]]]]]);
```

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
```




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

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

- ⦿ Generics

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

RegExp Methods

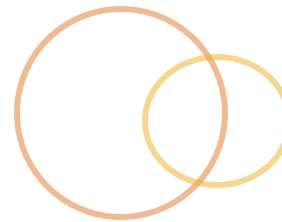
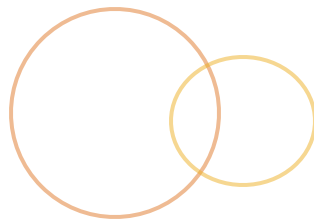


◎ Instance methods

```
re.exec(str)  
re.test(str)
```

◎ String methods that accept RegExp params

```
str.match(regex); // array of matches  
str.replace(regex, replacement);  
str.search(regex); // returns 1 at first match  
str.split(regex, limit); // returns array
```



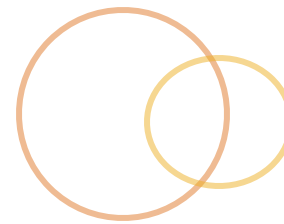
- ⦿ Error objects are thrown when runtime errors occur

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

- ⦿ Implementation varies across vendors
- ⦿ Instance properties

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

Error Handling



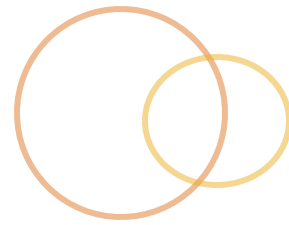
- ⦿ JavaScript is very lenient when it comes to handling errors
- ⦿ Internal errors are raised via the **throw** keyword, and are then considered "exceptions"
- ⦿ Exceptions are handled via a **try/catch/finally** construct, where the thrown exception is passed to the **catch** block
 - ⦿ Nesting allowed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

Error Throwing and Catching



```
function exceptionThrower() {  
    throw {  
        name: "ExceptionThrowerException",  
        message: "Bad things afoot"  
    };  
    //throw new Error("Bad things afoot");  
}  
  
try {  
    exceptionThrower();  
} catch (e) {  
    console.log(e);  
} finally {  
    console.log("Finally...");  
}
```

Built-in Errors



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

Exercise: Arrays



- 🕒 Open the following file:

`public/exercises/array/index.js`

- 🕒 Complete the exercise

- 🕒 Run the tests by visiting in your browser:

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

Solutions:

<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/array>

Exercise: Strings



- 🕒 Open the following file:

`public/exercises/string/index.js`

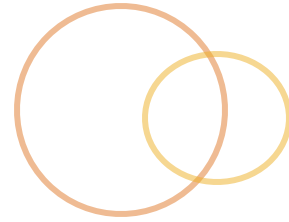
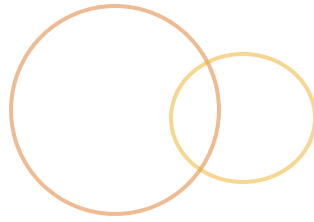
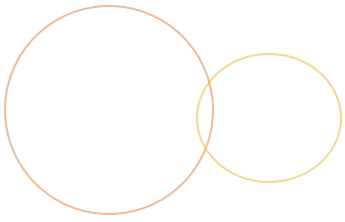
- 🕒 Complete the exercise

- 🕒 Run the tests by visiting in your browser:

`http://localhost:3000/exercises/string/`

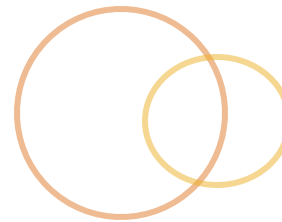
Solutions:

<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/string>



module

FUNCTION PATTERNS



⦿ Immediately Invoked Function Expression

⦿ A function that is defined within a parenthesis, and immediately executed

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

IIFE Uses



- 🕒 Define namespaces/modules/packages
- 🕒 Creates a scope for private variables/functions
- 🕒 Extremely common in JS

Privacy and modules with IFEs



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

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

Privacy and modules with IFEs



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

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!

Closure Example



```
function outer() {  
  let a = 1;  
  return function close_over_outer() {  
    console.log(a);  
    a++;  
  };  
};  
const witness = outer();  
witness(); // 1  
witness(); // 2  
witness(); // 3
```

Closure Module Example



```
const helper = (function() {  
  let secret = "I am special";  
  
  return {  
    secret: secret,  
    tellYourSecret: function() {  
      console.log(secret);  
    }  
  }  
})();
```

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


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



```
const 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');
```

Exercise: What's wrong here?



```
// given an integer representing a month
// return the month abbreviated name
const monthName = function(n) {

    const names = ["jan", "feb", "mar", "apr", "may",
"jun", "jul", "aug", "sep", "oct", "nov", "dec"];

    return names[n] || "";

}
```

Lazy Function Definition



```
const monthName = function(n) {  
  
  const names = ["jan", "feb", "mar", ...];  
  
  // we are re-assigning the var to a new fn!  
  // the new function will behave as a closure  
  monthName = function(n) {  
    return names[n] || "";  
  }  
  
  return monthName(n);  
  
}
```

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

Exercise: Closures - Guessing Machine

- 🕒 Open the following file:

`public/exercises/guessing-machine/index.js`

- 🕒 Complete the exercise

- 🕒 Run the tests by visiting in your browser:

`http://localhost:3000/exercises/guessing-machine/`

Solutions:

<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/guessing-mach>

Exercise: Closures - Temp Storage

- 🕒 Open the following file:

`public/exercises/closure/index.js`

- 🕒 Complete the exercise

- 🕒 Run the tests by visiting in your browser:

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

Solutions:

<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/closure>

Exercise: Closures - Hosts Module



- 🕒 Open the following file:

`public/exercises/hosts/index.js`

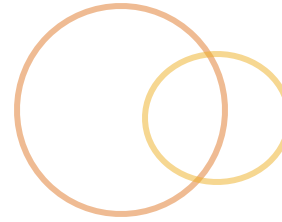
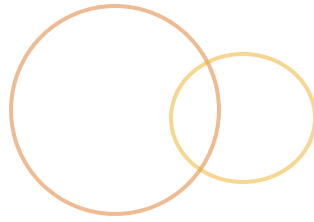
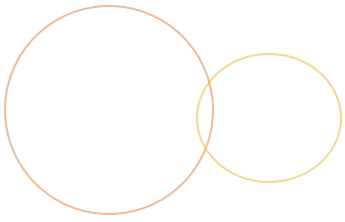
- 🕒 Complete the exercise

- 🕒 Run the tests by visiting in your browser:

`http://localhost:3000/exercises/hosts/`

Solutions:

<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/hosts>



module

OBJECT ORIENTED JAVASCRIPT

OO JS - Object Creation in JavaScript

- ◎ There's no "one" way in JavaScript
 - ◎ A rabbit hole of approaches
 - ◎ 4 competing JS engines, a lot of compromise in the definition of the language
- ◎ Lot's of people trying to emulate classical styles
 - ◎ Your soul *may* want JS to be like other OO-approaches
- ◎ Resist the urge to say, "where's my classes" ...
 - ◎ Accept that there is "no right way" ...
 - ◎ Learn about the many ways to create objects...
 - ◎ *Then decide which way to go with your team*

Object Creation in JavaScript



🕒 Object literal

🕒 `var me = {name: "Tim"};`

🕒 `Object.create(personObj)`

🕒 `var me = Object.create(null);`

🕒 Constructors w/ `new`

🕒 `var me = new Person("Tim");`

🕒 Factory Functions

🕒 `var me = makePerson({name: "Tim"});`

🕒 ES6 `class` keyword

🕒 `var me = new Person("Tim");`

Let's begin the OO Journey



- ◎ We create objects that represent the *things* of our system
 - ◎ They have methods for behavior
 - ◎ And properties for data
 - ◎ ...
- ◎ *What's something we want to work with?*
 - ◎ Animals
 - ◎ Vehicles
 - ◎ Washing Machines?

The Object Literal

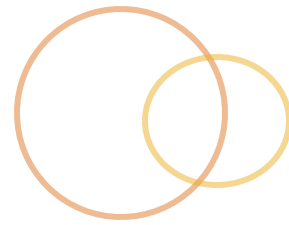


```
// We create Objects to represent Things in our  
// system, each with methods and properties
```

```
const dog = {  
  talk: function() {  
    console.log("Bark!");  
  }  
}
```

```
const cat = {  
  hasAttitude: true,  
  talk: function() {  
    console.log("Meow!");  
  }  
}
```

Prototypal Inheritance



```
// abstracting out shared behavior
```

```
const animal = {  
  talk: function() {  
    console.log(this.sound + "!!");  
  }  
}
```

```
// create an object with animal as it's prototype
```

```
const dog = Object.create(animal);  
dog.sound = "bark";
```

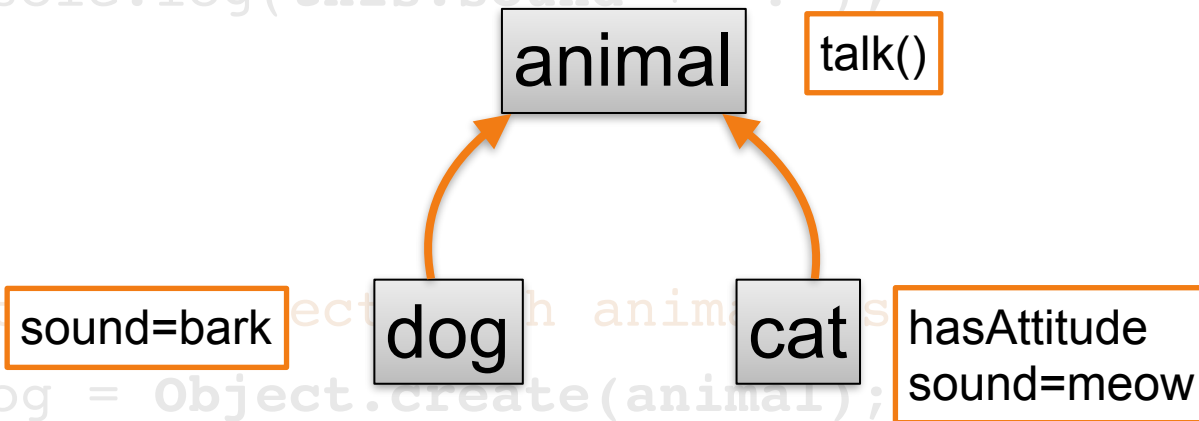
```
const cat = Object.create(animal);  
cat.hasAttitude = true;  
cat.sound = "meow";
```

Prototypal Inheritance



```
// abstracting out shared behavior
```

```
const animal = {  
  talk: function() {  
    console.log(this.sound + "!");  
  }  
}
```



```
// create each animal's prototype  
const dog = Object.create(animal);  
dog.sound = "bark";
```

```
const cat = Object.create(animal);  
cat.hasAttitude = true;  
cat.sound = "meow";
```

Prototype

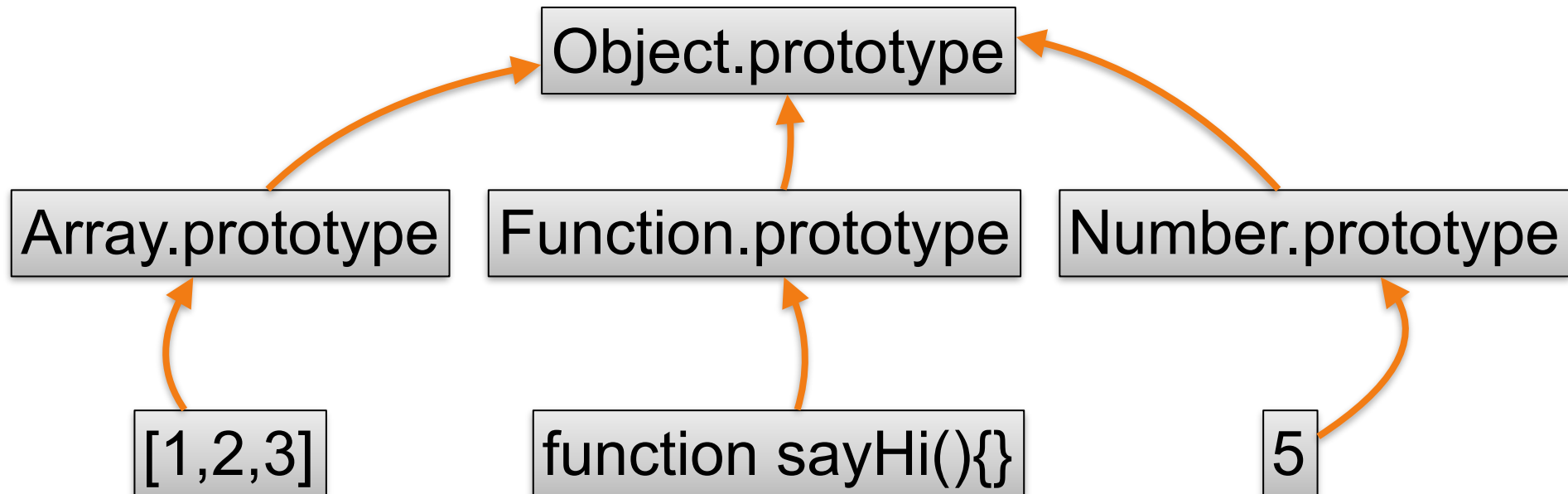


- ◎ **Prototype** – "an original or first model of something from which other forms are copied or developed"
- ◎ Objects have an internal link to another object called its *prototype*
- ◎ Each prototype has its own prototype, and so on, up the ***prototype chain***
- ◎ Objects ***delegate*** to other objects through this prototype linkage
 - ◎ "For this object, use this other object as my delegate"

Built-in Objects



- Built-in JS objects use prototypal inheritance and prototype objects (`__proto__` vs `prototype`)
- Array, Number, etc... store *generic* methods
- Array.prototype, etc, store *inherited* methods



.prototype vs. __proto__



- ◎ **.prototype** is a property of the Function object
 - ◎ Every Function object has one
 - ◎ When a function is used as a constructor, new objects will point to **.prototype** as their "prototype"
 - ◎ *"When I create an Array instance, it delegates to Array.prototype"*
- ◎ **__proto__** is an instance property of an object
 - ◎ References its "prototype"
 - ◎ Prototype Chain
 - ◎ *"When I create an Array instance, use an internal property __proto__ to point to Array.prototype"*
 - ◎ Not standard until ES6

Prototype Augmentation



- 🕒 The linkage is live, you can extend at run-time and affect all copies

```
const animal = {};  
  
const dog = Object.create(animal);  
  
// setting a property on the prototype of dog  
animal.hasTail = true;  
  
console.log(dog.hasTail); // ?
```

Constructors and new



- ⦿ A function that expects to be used with the ***new*** operator is said to be a constructor

```
const MyConstructor = function(name) {  
  // set instance-level properties  
  this.name = name;  
}  
  
// set delegated methods and properties...  
MyConstructor.prototype.sayHello = function() {};  
  
const instance = new MyConstructor("DogCat");
```

Pseudo-Classical Inheritance



```
// We create a function to serve as our constructor
// which sets instance properties
const Animal = function (sound) {
  this.sound = sound;
}

// We use it's prototype to define delegated props
Animal.prototype = {
  talk: function() {
    console.log(this.sound + "!");
  }
}

const dog = new Animal("bark");
const cat = new Animal("meow");
cat.hasAttitude = true;
```

Pseudo-Classical Inheritance



```
// We create a function to serve as our constructor
// which sets instance properties
```

```
const Animal = function (sound) {
  this.sound =
```

Animal

.prototype

talk()

```
// We use it's prototype to define delegated props
```

```
Animal.prototype = {
  talk: function() {
    console.log(`I am a ${this.sound}!`);
  }
}
```

sound=bark

dog

cat

hasAttitude
sound=meow

```
const dog = new Animal("bark");
const cat = new Animal("meow");
cat.hasAttitude = true;
```

Constructors and Inheritance



- ⦿ Depends on usage of **new** keyword, constructor functions and the prototype linkage
- ⦿ Still... isn't like classes
- ⦿ Only supports single-inheritance
- ⦿ Since inheritance is programmatic in JavaScript, we can create helpers to make things easier:
 - ⦿ <http://jsfiddle.net/jmcneese/p2ohmuw0>

Pseudo Classical continued



```
const Animal = function (sound) {  
  this.sound = sound;  
}
```

```
Animal.prototype = {talk: function() {}}
```

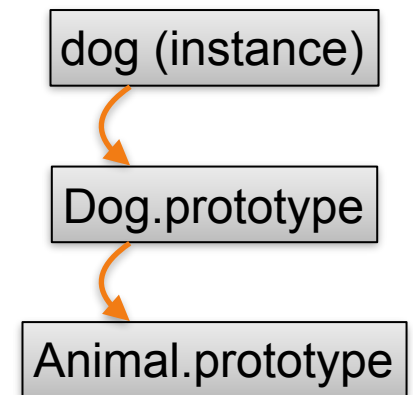
```
const Dog = function(breed) {  
  // apply the superclass constructor  
  Animal.call(this, "bark");  
  this.breed = breed;  
}
```

```
// Dog extends Animal
```

```
Dog.prototype = Object.create(Animal.prototype);  
Dog.prototype.wag = function() {};  
Dog.prototype.constructor = Dog; // we overwrite this
```

```
const doggy = new Dog("Robot");
```

We want a prototype chain of:





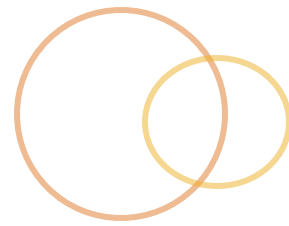
Pseudo Classical continued

```
const Animal = {  
  this.sound = 'bark';  
  talk() {  
    console.log('bark');  
  }  
};  
  
Animal.prototype = {talk: function() {}};  
  
const Dog = function(breed) {  
  // apply the class prototype to this instance  
  Animal.call(this, 'bark');  
  this.breed = breed;  
};  
  
// Dog extends Animal  
Dog.prototype = Object.create(Animal.prototype);  
Dog.prototype.sound = 'bark';  
Dog.prototype.breed = 'Robot';  
// we overwrite this  
  
const doggy = new Dog("Robot");
```

Diagram illustrating the prototype chain for the `Dog` constructor:

- `Animal` (constructor) has a `talk()` method and a `prototype` object.
- `Dog` (constructor) has a `wag()` method and a `prototype` object.
- `doggy` (instance) has `sound=bark` and `breed=Robot` properties.
- The `prototype` of `Dog` is `Object.create(Animal.prototype)`, meaning it inherits from `Animal.prototype`.

Setting the prototype



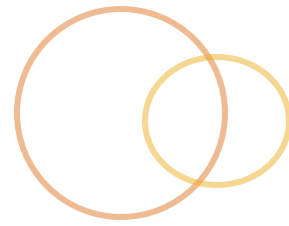
```
//slow
child.__proto__ = parent;

// class-like w/ constructors
MyFunction.prototype = parent;
let child = new MyFunction();

// slow, should be avoided
Object.setPrototypeOf(child, parent);

// fav
let child = Object.create(parent);
```

Reading the prototype



```
// check instance of  
[1, 2, 3] instanceof Array;  
  
// check prototype of  
String.prototype.isPrototypeOf([1, 2, 3]);  
  
// get prototype of  
Object.getPrototypeOf(child);  
  
// not widely supported  
child.__proto__;
```

Prototype vs Class



- JavaScript leverages **prototypal inheritance** instead of **class-based** inheritance

- Classes...

 - Act as blueprints

 - You make copies

- Prototypes...

 - Act as delegates

 - Live representative, not a copy

- ES6 `class` keyword

 - Just a wrapper around prototype, so... `_(ツ)_/`

Gotcha - Be careful



```
function Animal(name) {
  this.name = name;
}

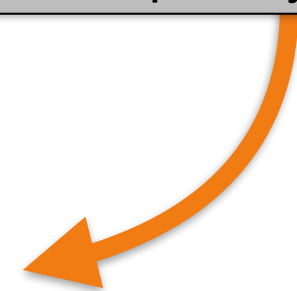
Animal.prototype.walk = function() {
  alert(this.name + ' walks');
};

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = Animal.prototype;
Rabbit.prototype = Object.create(Animal.prototype);

Rabbit.prototype.walk = function() {
  alert(this.name + " bounces!");
};
```

You likely want Rabbit's
prototype to delegate
Animal's prototype



Factory Function Pattern



- ◎ Functions that create and return objects
- ◎ Alternative to constructors
- ◎ Better encapsulation & privacy
- ◎ Retains context (through closures)

Factory Function Example



```
function dogMaker() {  
  const sound = "woof";  
  return {  
    talk: function() {  
      console.log(sound);  
    }  
  }  
}
```

```
const dog = dogMaker();  
dog.talk();
```

```
// real-world practical bonus here  
// this retains context and works!  
setTimeout(dog.talk, 1000);
```

Object Composition



⦿ When objects are *composed* by *what it does*, not *what it is*

⦿ Animal

-> Cat

-> Dog

VS

⦿ Animal

-> Animal + Meower

-> Animal + Barker

⦿ Alternative to multiple inheritance

⦿ Properties from multiple objects are copied onto the target object

Functional Composition Example

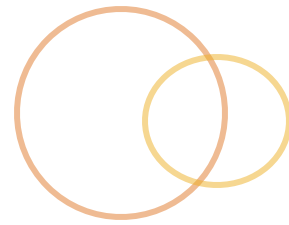


```
const animal = {legs: 4}

const meower = function (obj) {
  this.sound = "Meow";
  this.purr = function() {}
}
const barker = function () {
  this.sound = "Bark";
}

const cat = meower(animal);
const dog = barker(animal);
// And this is easier w/ Composition
const dogCat = barker(cat);
```

Mixins Example



```
function CatDog() {  
  Dog.call(this);  
  Cat.call(this);  
}
```

```
// inherit one class
```

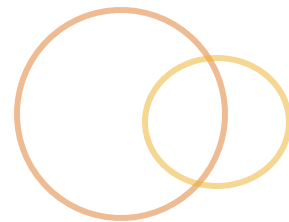
```
CatDog.prototype = Object.create(Dog.prototype);
```

```
// mixin another
```

```
// Object.assign is ES6 object merging)
```

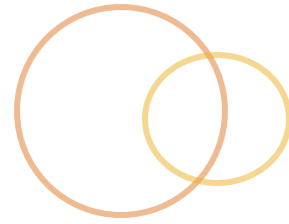
```
Object.assign(CatDog.prototype, Cat.prototype);
```

Class keyword [ES6]



- ⦿ Just syntactic sugar over prototypes
- ⦿ Leaky abstraction; you'll still deal with prototypes
- ⦿ Not hoisted (like function declarations are)
- ⦿ Uses `class` & `constructor` keywords

Pre-class keyword



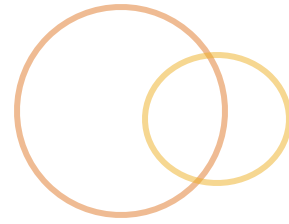
Without class

```
const Human = function(name) {  
  this.name = name;  
}
```

```
Human.prototype.talk = function(str) {  
  console.log(this.name, "says", str);  
}
```

```
let tim = new Human("tim");  
tim.talk("Hi!");
```

Class keyword [ES6]



With class

```
class Human {  
  constructor (name) {  
    this.name = name;  
  }  
  
  talk(str) {  
    console.log(this.name, "says", str);  
  }  
}
```

class and **constructor** keywords

abbreviated method properties

```
let tim = new Human("tim");  
tim.talk("Hi!");
```

But you are still using **new**, **this** and **prototype**

Extending Classes

```
const Rectangle = class {  
  constructor(width, height) {  
    this.width = width;  
    this.height = height;  
  }  
  get area() {  
    return this.width * this.height;  
  }  
}
```

no literal properties allowed here :(

```
class Square extends Rectangle {  
  constructor (width, color) {  
    super(width, width);  
    this.color = color;  
  }  
  someMethod() {  
    return "Hi";  
  }  
}
```

Class keyword extras



- ◎ You can **extend** traditional function-based "classes"
- ◎ Can define **static** methods
 - ◎ Won't be created on instances
- ◎ Can define **getters** and **setters** with get and set method keywords

OO – Recap



- ◎ No classes, only prototypes
 - ◎ Prototypes are full-fledged objects that new objects use to delegate behavior to
 - ◎ Everything derives from Object
- ◎ Fundamental concepts are fully supported
- ◎ Encapsulation/visibility can be implemented via closure/IIFE patterns
- ◎ Objects and their properties are runtime configurable
 - ◎ As are their mutability settings
 - ◎ Enough rope to hang yourself with, so be careful!

OO – Exercise



- ◎ **Create a hierarchy of objects**
 - ◎ Cats, Dogs, Animals
 - ◎ Me, People, Mammals
 - ◎ Car, Truck, Vehicles
- ◎ First using just `Object.create()`
- ◎ Then with constructors and/or the `class` keyword

Exercise: Object Oriented Design in JS



- ◎ First - let's agree on a entity hierarchy

 - ◎ Animals, Trees, Washing Machines?

- ◎ Open the following file:

 - `public/exercises/ood/index.js`

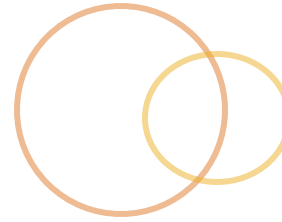
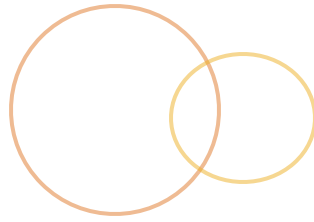
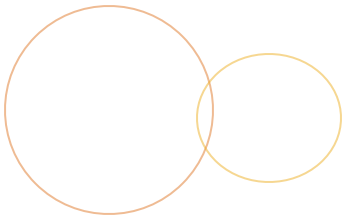
- ◎ Complete the exercise

- ◎ Debug in the console:

 - `http://localhost:3000/exercises/ood/`

Solutions:

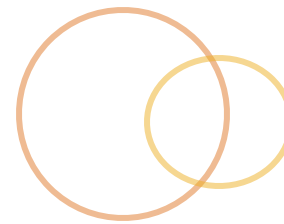
<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/ood>



the end is near

WRAPPING UP

Exercise: Stubs



- 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
- Open the following file:
`public/exercises/stub/index.js`
- Complete the exercise
- Run the tests by visiting in your browser:
`http://localhost:3000/exercises/stub/`

Solutions:

<https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/stub>