learning spike

Core JavaScript

Ryan Morris
@mrmorris

Introductions

- About me…
- About you…
  - What do you do?
  - What is your programming background?
    - Any front-end?
    - 😍, 😡 or 😭 JavaScript?
  - What do you hope to gain from this course?
- What is your current development environment and process?

How the class works

- Mixture of labs and lecture
- Informal
  - Stop me anytime
  - Discussion > Lecture
  - Outline is flexible
    - **There is too much to cover** so we'll adjust as needed
- You'll help define areas of focus
- Class assessment towards 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

- Data types & variables
- Object basics
- Arrays
- Control Flow
- Functions
- Hoisting
- Lexical vs Dynamic Scope
- Closures

I wasn't planning to cover

- Objects in depth
- OO, new keyword
- ES6 in depth
- Modules

~Mostly ES5~

~Mostly for beginners~

~let's shape it~

Resources

- Reading List
  - https://javascript.info/intro
- Documentation
  - http://devdocs.io
  - https://developer.mozilla.org/en-US/docs/Web
  - http://kapeli.com/dash (Mac only)
  - Google it.
- Compatibility checks
  - http://caniuse.com

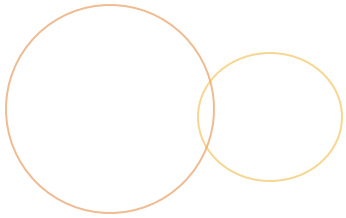
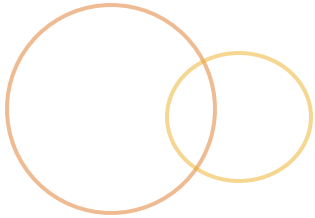
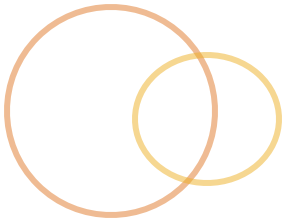
Lab prep - set up our toolkit

- A browser with dev tools
  - Preference for Chrome in class
  - Open your browser and hit F12 or alt/opt/⌥ -⌘-i
- Our web editor, jsfiddle
  - Does this work:
    - http://jsfiddle.net/mrmorris/8wfu5tct/
  - Sign Up
    - http://jsfiddle.net/

👆 Everyone OK with the above?

PDF for today

- In case you didn't quite catch something, or can't see the screen well:
  - Head to my repository
    - ###
  - Go to “**/docs**” folder
  - Or just download the whole repo…


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 as "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?

- Despite the shortcomings, it's pretty awesome
  - Very expressive
  - Very flexible (that multi-paradigm thing)
  - Lightweight
- The language of the web
  - Integrates nicely w/ HTML/CSS
  - Supported across all browsers
  - Simple to use
  - A server and command line services
- Beginning to dominate the entire software stack
- *Easy to learn, hard to master*

Approaching JavaScript

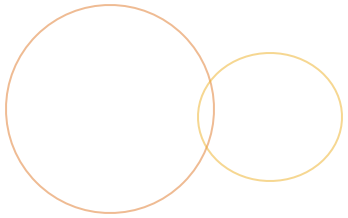
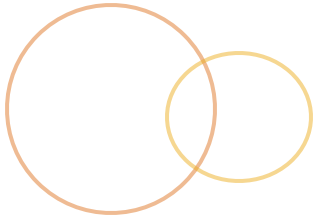
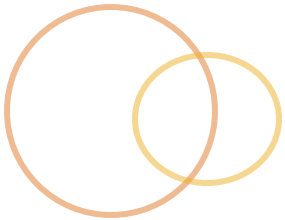
- It's *not* Java
- It's *not* class-based
- Very dynamic
- Supports imperative, functional and object-oriented approaches

Where does JavaScript live?

- Plain text files, not compiled
  - *Though this is changing*
- Browser (Built-in Engine)
  - Inline <script> blocks
  - Linked <script src="file.js"> files
- Server (Node)
  - One script file
  - Set of modules

Languages on 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!

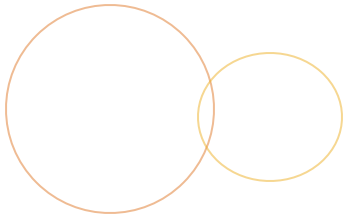
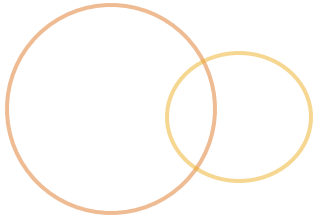
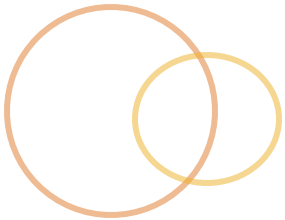





module

WARM UP

Pop quiz!

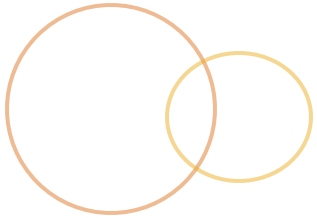
- **How well does everyone know programming constructs**
  - if-else
  - for loops
  - functions
- **Core JS concepts?**
  - Coercion
  - Hoisting
  - Scope
  - Context
  - Functions
  - Prototype

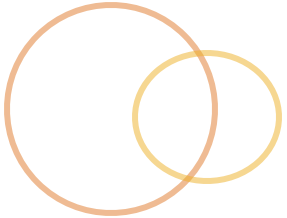





obligatory

HELLO WORLD

Say 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

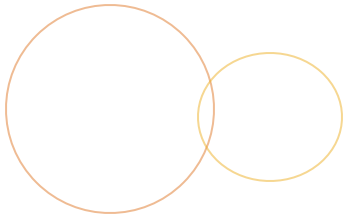
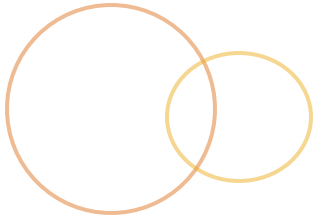
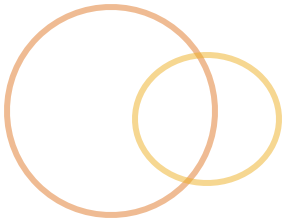
Log hello

- Now try

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

Browser Console

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


module

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
  - `var x = 1 + 2;`
- They are *mostly* optional
  - Automatically inserted but not fail-safe
  - So, don't rely on it…

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

<= results in TypeError

Comments

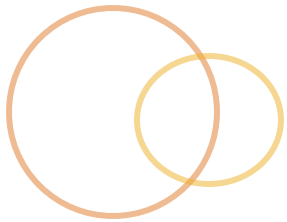
- Follow C/C++ conventions
- Multiline

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

- Single line

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

Declaring variables



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

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

- Or in sequence:

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

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

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

- Default value will be `undefined`

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

Declaring variables [ES6]

- **let** & **const**
  - block scoped
  - can’t redeclare
  - let is mutable while const is immutable*

```
let x = 0;
let y = {};

x = 5; // ok!
let x = 5; // TypeError

const z = 5;
const u; // SyntaxError
z = 10; // ReferenceError
```

Variable names

- Only two limitations
  - Contains letters, digits, _, or $
  - Can’t begin with a digit
  - No reserved keywords
- CaSE matters
- Unicode characters are supported

```
var 🍔 = 'burger';
```

Variable scope

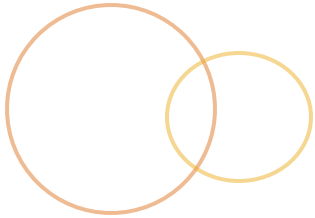
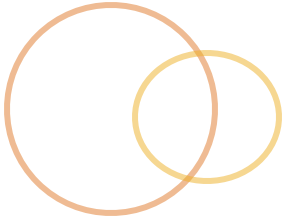
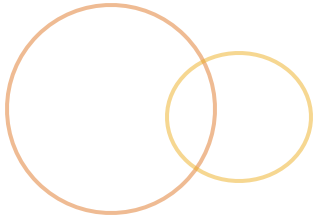
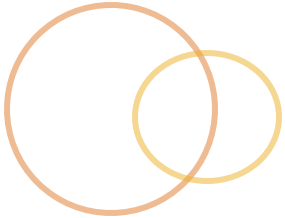
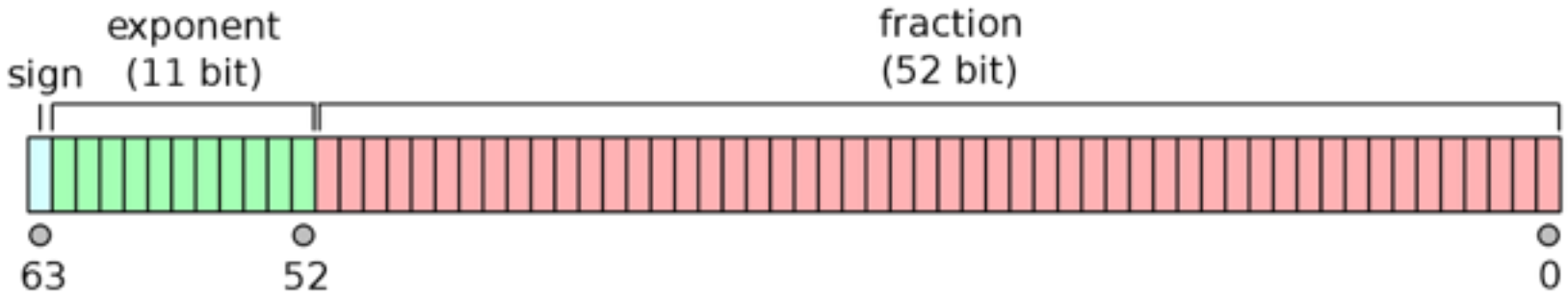
- JavaScript is **function-scoped**
  - **var** is used to define variable in current function's scope
  - variable is said to be "local" to a function when defined within it
- … supports **block-scoped** in [ES6]+
  - **let**, **const**

```
if (expression) {
    let x=1; // scoped to this block only
}
console.log(x); // Temporal Dead Zone error
```

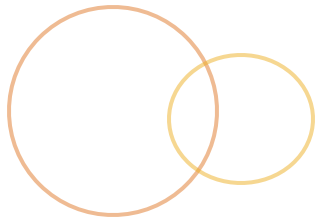
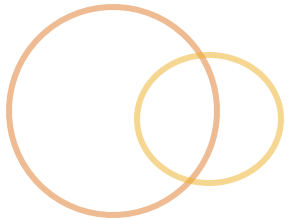
Data Types

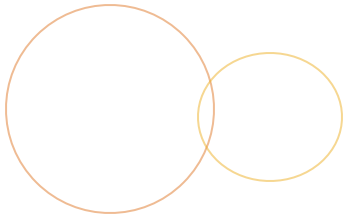
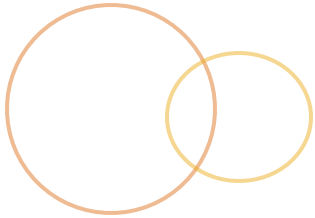
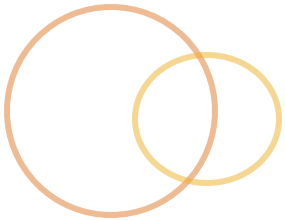
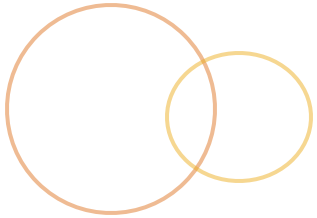
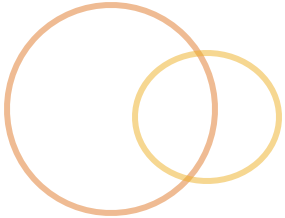
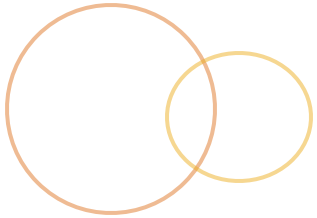
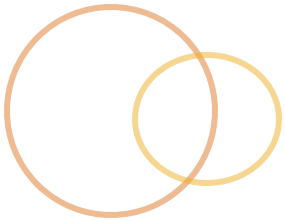
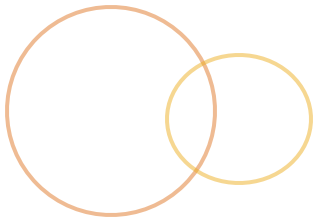
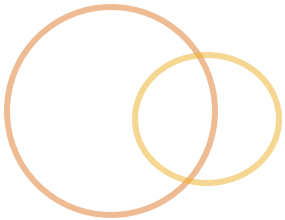
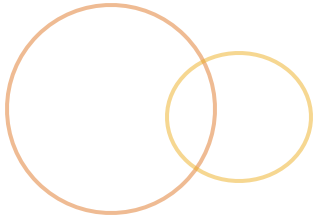
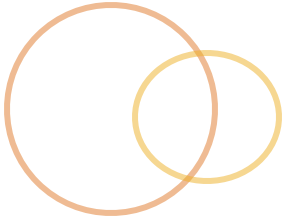
- Five *primitive* data types:
  - strings
  - numbers
  - booleans
  - null - lack of value
  - undefined – no value set, the default in JavaScript
  - *ES6: Additional primitive, Symbol*
- And then Objects
  - Property names referencing values
  - ie: Object, Array, Function, Math…
  - Function is a callable object
- ES6: Adds the Symbol to create identifiers

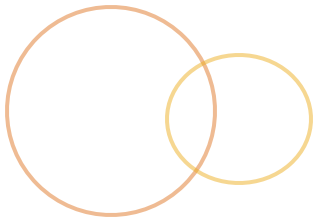
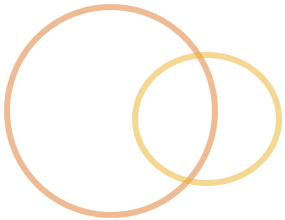
undefined & null

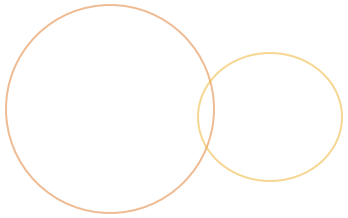
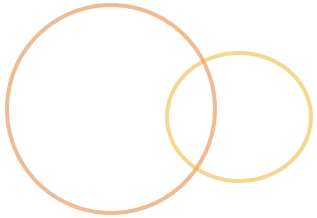
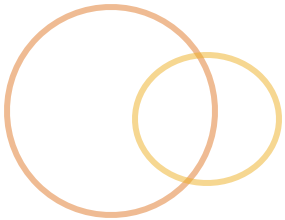
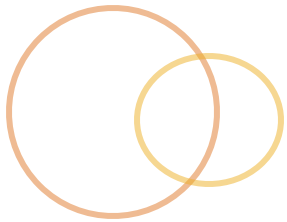
- Little difference between the two, in practice
- Variables declared without a value will start with `undefined`
- Can compare to `undefined` to see if a variable has a value
  - ```
    var a;
    a === undefined;       // true
    typeof a;          // undefined
    ```
    ```

boolean





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

number






- 64bit floating point
- Numbers can be expressed as:
  - Decimal: **-9.81**
  - Scientific: **`2.998e8`**
  - Hexadecimal: **`0xFF;`** `// 255`
  - Octal: **`0777`**
  - `Binary:` **`0b10000000000000000000000000000000`**
- Special numbers:
  - `NaN`
    - `0/0; // NaN`
    - `NaN == NaN; // false`
  - `Infinity and -Infinity`
    - `5/0; // Infinity`

number issues

- All numbers are stored internally as 64bit floating points



- Integers are accurate up to 15 digits

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

- Decimals are accurate up to 17 digits

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

- What to do?
  - Round, use only integers
  - Use libs like **BigDecimal** or **Big.js**

string

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

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

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

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

{object}

- A list of **key:value** pairs, surrounded by **curly braces**
  - Considered a *Dictionary*, *Hash* or *Map* in other languages
- Keys:
  - Unordered
  - Must be strings
    - Don’t require quotations unless if they contain special characters
- Values:
  - can be any type of data, including functions

```
var obj = {bar: “baz"};

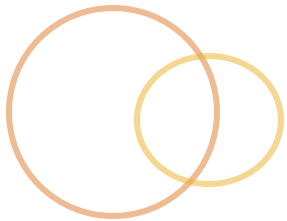
obj.foo = true; // assign values
obj.bar;   // dot-accessor
obj['foo'] // array-accessor
```

[arrays]

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

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

- Strange behavior if you try to use string keys

```
var arr = [1,2,3];
arr.length; // 3
arr['bar'] = 10;
arr.length; // 3
```

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”

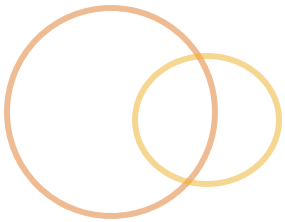
typeof(0);        // also ok
```

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/

typeof and NaN

- Wha?
  - `typeof NaN; // “number” <– huh?`
  - `NaN === NaN; // false  <– bummer…`
- isNaN(); is NaN… not “is not a number”
  - *coerces via Number() constructor*
  - isNaN(NaN); // true
  - isNaN(‘’); // false
  - isNaN(5); // false
  - isNaN(‘123ABC’); // true
  - isNaN({}); // true
- Number.isNaN()
  - does not coerce

Type Coercion

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

```
+"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
  - `[] + [] -> empty string`
  - `[] + {} -> [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

Exercise - typeof an Array?

- What is the output of:

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

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…

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

typeof objects

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

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

- *except **Functions**

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

Everything* is an object

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

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

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

Literals





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

```
5           // number literal
"a"         // string literal
true        // boolean literal

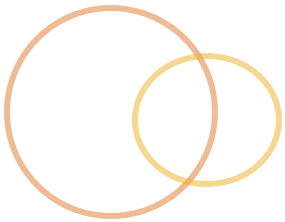
{}          // object literal
[]          // array literal
/^(.*)$/ // regexp literal
```

Literals by Construction

- Literals constructed by their object counterpart

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

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 - Variables Round 2

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






module

OPERATORS

Unary





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

Arithmetic

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

Bitwise





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

Assignment

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

Relational





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

Equality* (strict vs loose)

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

Logical





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

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

Ternary





```
// condition ? then : else;

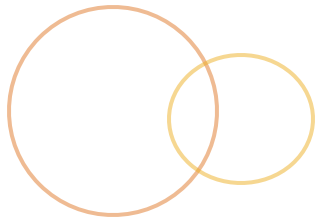
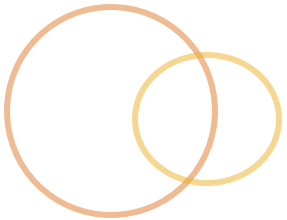
true ? 'foo' : 'bar' // 'foo'
```

Falsy / Truthy

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

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();
```

Exercise - Truthing and Falling

- Two things to ponder:
  - Is the expression **truthy or falsy**
  - and 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.[]
```






module

CONTROL STRUCTURES

Conditionals & Loops

Control Structure Examples

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

Conditional statements

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

```
    …
} else {
    …
}
```

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

Switch statements



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

    default:
        // statements
        break;
}
```

Basic Loops



- `for`
- `while`
- `do…while`

looping - for

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

looping - while

```
var i = 0;

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

looping - do/while



```
var i = 0;

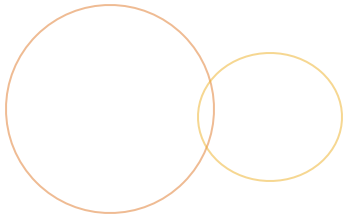
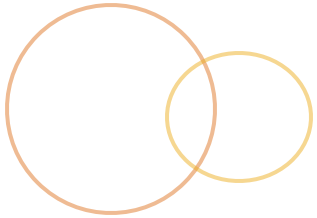
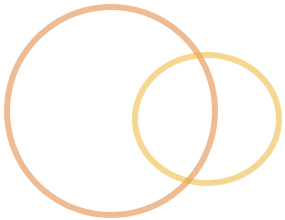
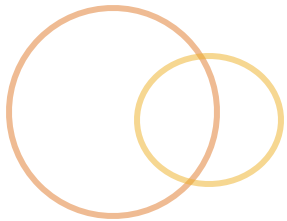
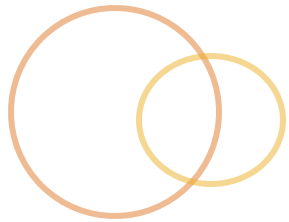
do {
     // do stuff 10 times
     i++;
} while (i < 10);
```

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);
}
```

looping through an array

```
var cats = [1,2,3];

for (var i=0; i < cats.length; i++) {
  console.log(cats[i]);
}


// in ES6
cats.forEach(function(el, i, arr){
  console.log(el);
});
```

Enumerating over objects

- `for…in`
  - Over object properties
- `for…of (ES6)`
  - Over any Iterable
- ~~`for each…in`~~
  - `deprecated, looping 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
```

Control Structures Recap

- Conditionals like if and if-else
- Switch statements
- Iterate (loop) with while and for
- Enumerate over an object with for..in

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/






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*
- **Callable objects**

Defining a function



- Four ways
  - Function **declaration**
  - Function **expression**
  - `Function()` **constructor**
  - [ES6] **Fat arrow**
- 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;
}
```

- 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 addComm(text, 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 doSomething() {
  // 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);

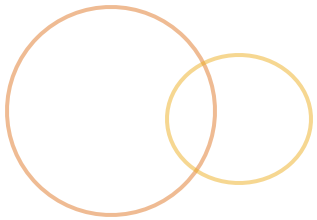
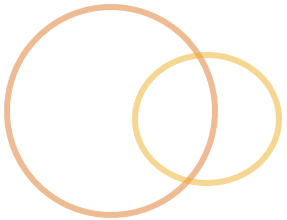
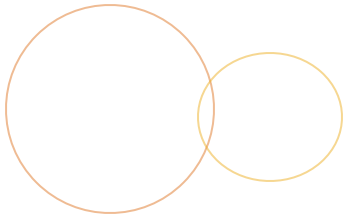
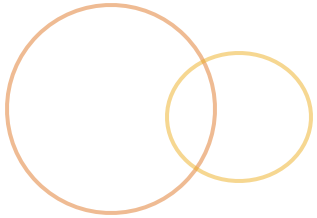
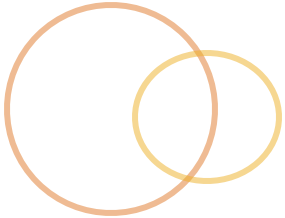
  console.log(args);
}

doSomething(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

this?

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

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

runMe(); // ?
```

`this` is context

- Available on invokation
- Refers to the context of the function at call-time
  - Dynamically bound (not lexical)
  - *"The object in context that invoked the method"*
- Basis of
  - Inheritance
  - Multi-purpose functions
  - Method awareness of their objects
- *We'll come back to* ***`this`***

this example

```
var person = {
  name: “John Doe”,
  speak: function() {
    console.log(“Hi my name is”, 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(); // ?
```

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

Timers





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

Exercise – Functional FizzBuzz

- **FizzBuzz Function**

  Create a function that:
  - Accepts a single number argument
  - Returns the proper FizzBuzz result for that number
  - Loop through numbers 1-100 to test your results
  - BONUS: Write tests!

- 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/






module

SCOPE & HOISTING

Scope

- Variable **access** and **visibility** in a piece of code at a given time
  - `var` declares a variable within the current scope
- **Lexical scoping** (static)
  - as opposed to *d*ynamic
  - Scope is defined at author-time
  - No need to execute; you can read code and determine scope
- **Function scope**
  - Functions are the only thing that can create a new scope
  - A variable declared (with var) in a function is visible only in that function and its inner functions. But not the other way around.
- ES6 supports **block scope**, *we'll come back to that*

Function Scope

- What are the scopes here?

```
var a = 5;
function foo(b) {
  var c = 10;
  d = 15; // no var! globally scoped

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

- Each *inner* function has access to the *outer scopes*
  - outer function scope cannot access the inners
- **ReferenceError** indicates unknown variable in scope

Block scope

- I lied…
  - eval() can cheat scoping, inserting itself into the scope
    - But… in *strict mode* eval() creates its own scope
  - An exception's "catch" block has its own scope
- And in an ES6 world..
  - **let** & **const** define variables within any block's scope
  - but same visibility rules apply

```
let x = 5;

if (x > 1) {
  let x = 10; // OK, shadows outer x
  let y = 20;
}

console.log(x); // 5
console.log(y); // ReferenceError - not defined
```

The Global Scope

- Variables defined w/out “var” are defined in the global scope

```
alert() === window.alert(); // true
var user_id = 5;
window.user_id; // 5
```

- “use strict” will prevent you from creating a global var
- let & const do not define variables in the global object

```
let user_id = 5;
window.user_id; // undefined
```

- Craft your scope
  - Don’t pollute global
  - Use functions and closures to define scopes and conceal vars

Exercise: Hoisting (pt 1 of 3)

- What will the output be?

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

 console.log(x); // ?
 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);
 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);
 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();

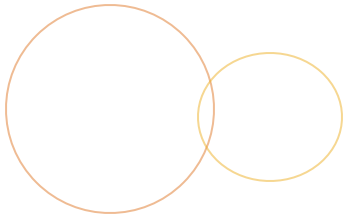
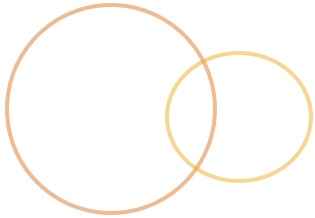
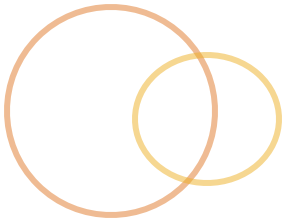

function foo() {
 console.log("Foo!");
}


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

Becomes…

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


foo();
bar();


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;
```

Becomes…

```
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’);
}
```

Exercise – Function Scope

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






module

OBJECTS

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

Objects

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

Creating an object literal

- Create an object literal with {}:

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

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

Object properties

- Dot Syntax
  - myObj.key;
- Square bracket Syntax
  - myObj[“key”];
- Can delete a property with `delete`
  - `delete myObj.key;`

Properties descriptors

- Object properties have descriptors that affect its behavior
  - `enumerable`
  - `configurable`
- `Object.defineProperty(obj, “key”, {`

```
    .. descriptors ..
}
```

Object reflection

- Objects actually “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
- `in` operator
  - `“propertyName” in object`
  - Goes all the way up the chain, not just “own”
- `hasOwnProperty`
  - `myObj.hasOwnProperty(“propertyName”)`

Object enumerating

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

Object enumeration, continued

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

Exercise - Mutations

- What will the result of this be:

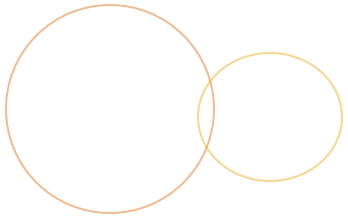
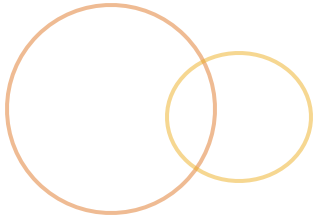
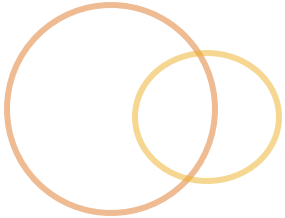
```
var rabbit = {name: 'Tim'};
var hp = 100;

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

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

Mutability

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

Exercise: Copy Object

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


module

BUILT-IN OBJECTS

Lot's of Built-in Objects

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

String

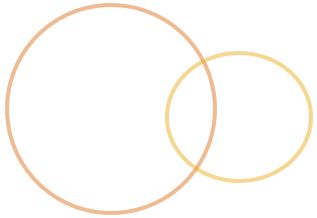
- Instance properties

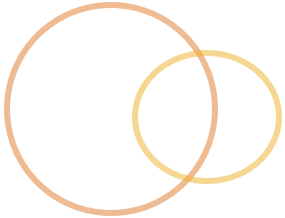
```
new String('foo').length // 3
```

- Instance method examples

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

Number





- Generics

  `Number.MIN_VALUE`

  `Number.MAX_VALUE`

  `Number.NaN`

  `Number.POSITIVE_INFINITY`

  `Number.NEGATIVE_INFINTY`

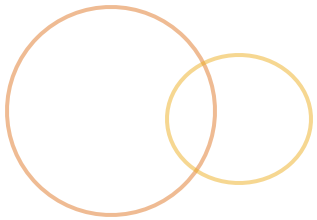
- Instance method examples

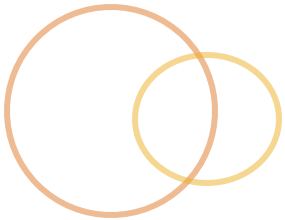
```
var num = new Number(3.1415);
num.toExponential();      // "3.1415e+0"
num.toFixed();            // 3
num.toPrecision(3);       // 3.14
```

Math

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

Array





```
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]
arr.concat([2, 4]);    // [1, 1, 2, 4]
arr.join('-');                 // "1-1"
arr.slice(1, 1);            // [1]
```

Date

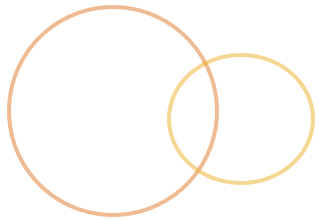
- Represents a single moment in time based on the number of milliseconds since 1 January, 1970 UTC
- Generics

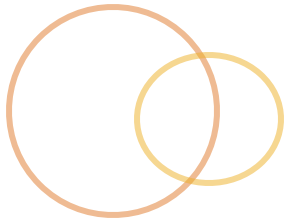
```
Date.now()
Date.parse('2015-01-01')
Date.UTC(2015, 0, 1)
```

- Instance method examples

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

RegExp





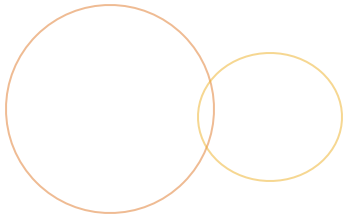
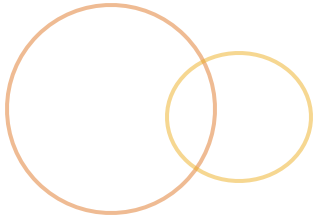
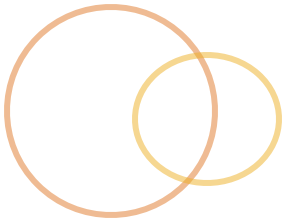
- Creates a regular expression object for matching text with a pattern

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

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

Exercise – Core objects

- **Reverse an array**
  Without array.reverse()
  - http://jsfiddle.net/mrmorris/zqfwy1xp/
- **Data Grids**
  Display an array of objects as a table in the console
  - http://jsfiddle.net/mrmorris/0kptbv7p/
- **Arrays**
  Use array methods like filter, map, reduce to search and mutate arrays
  - https://jsfiddle.net/mrmorris/ce7s09j0/
- **Strings**
  Replacing a word in a string
  - https://jsfiddle.net/mrmorris/owrtzequ/


module

CONTEXT

Scope & Context

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

“this” is the context

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

this example (again)

```
var person = {
  name: “John Doe”,
  speak: function() {
    console.log(“Hi my name is”, 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(); // ?
```

“this” is determined by call-site

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

“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

More explicit binding

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

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
  - Support super-short syntax in a few ways
  - No **`arguments`** of its own
    - it's the arguments of the outer function
  - No **`this`** (context) of its own
    - Lexically bound; wherever the function was defined

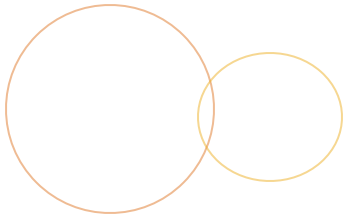
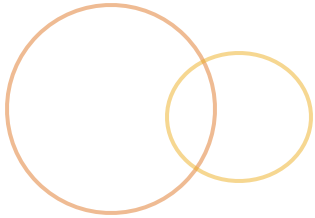
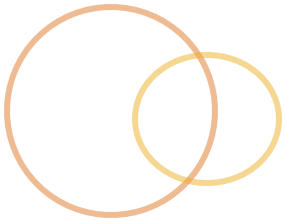
```
var add = function (x) {
  return x + 1;
}

// becomes
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?
}

// Gotcha…
me = {
  name: 'Tim',
  talk: (x) => {
    console.log(this.name, x);
  }
}
```

Exercise - Objects

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


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

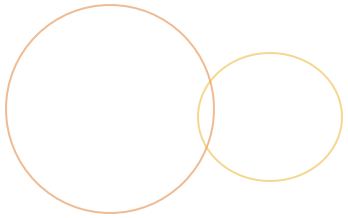
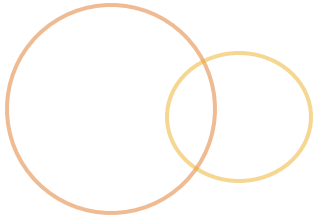
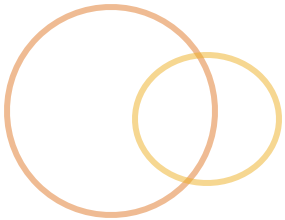
- Define namespaces/modules/packages
  - Typically singletons or “static” objects
- Creates a scope for private variables/functions
- Extremely common in JS

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

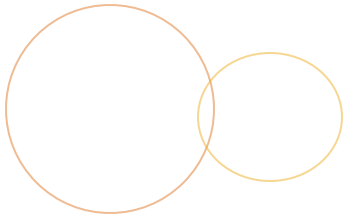
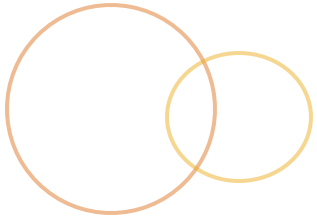
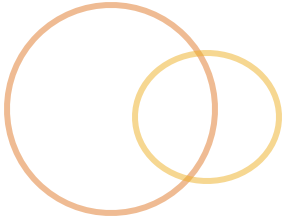
```
var helper = (function() {
  var secret = “I am special”;

  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/


module

FUNCTION PATTERNS

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

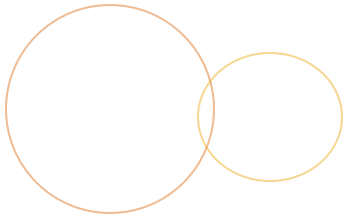
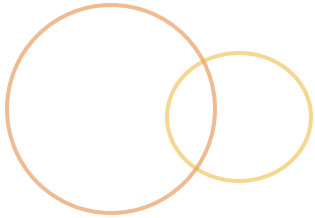
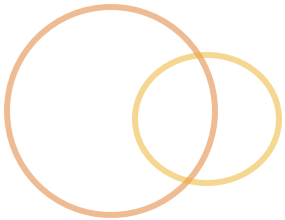
- Are **Objects** with their own methods and properties
- Can be **anonymous**
- Can be bound to a particular **context**, or particular **arguments**
- Can be **chained** together, provided the return of each function has methods
- **Closures** can be used to maintain access to calling context's variables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

Strict vs Sloppy

- `"use strict";`
- It kills deprecated and unsafe features
- It changes "silent errors" into thrown exceptions
- It disables features that are confusing or poorly thought out
  - Ensures "eval" has its own scope
  - Does not auto-declare variables at the global level during a scope-chain lookup
- Can be set globally or within function block
  - Careful when concatenating scripts
- http://jsfiddle.net/mrmorris/d2f6hohb/

Final Exercise

- **Track Temperatures**
  Build a mini module to store and track temp values
  - Fork:
    - 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/


the end is near

WRAPPING UP

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**