Practical JavaScript

Out of the console, into the real world!

Practical JavaScript

1. Intro to Objects
2. What is the DOM, anyway?
3. The DOM API
4. Example One: a simple calculator
5. Example Two: a "read more" link
6. Progressive Enhancement

1. Intro to Objects

Unlike people, it is often good to objectify JavaScript data

Intro to Objects

- Variables are great, but they only let us store one value
- If we have lots of related values to track, it gets tricky
- Imagine coding a JS app to track contacts and their info...

```
var contact1FirstName = 'Josh';
var contact1LastName = 'Collinsworth';
var contact1CellNumber = 5558675309;
var contact1HomeNumber = 4815162342;
var contact1Image = 'images/josh.jpg';
```

Intro to Objects

- Arrays are a great way to store lots of related values
- However, they also depend on us knowing where each piece of data is in the array

```
var contact1 = [
    'Josh',
    'Collinsworth',
    5558675309,
    4815162342,
    'images/josh.jpg'
];
```

Intro to Objects

- Objects give us a way to store lots of **related values** in a single, simple container!
- So instead of this...

```
var contact1FirstName = 'Josh';
var contact1LastName = 'Collinsworth';
var contact1CellNumber = 5558675309;
var contact1HomeNumber = 4815162342;
var contact1Image = 'images/josh.jpg';
```

Intro to Objects

- ...We can have this:

```
var contact1 = {

    'FirstName': 'Josh',
    'LastName': 'Collinsworth',
    'CellNumber': 5558675309,
    'HomeNumber': 4815162342,
    'Image': 'images/josh.jpg'

}
```

Intro to Objects

- Objects let us store a **collection** of properties and values
- Objects are essentially arrays, but with property/value pairs instead of a list of single items.
- You'll hear the phrase "object-oriented programming." That means working with objects such as these. It's a powerful and useful way to do things!

```
var myObject = {
        firstName: "Josh",
        lastName: "Collinsworth",
        age: 36,
        bearded: true
};
```

Arrays are essentially objects

- An array is basically an object, but with numbers instead of named keys.

```
var luckyCharms = [
    'Hearts',
    'Stars',
    'Moons',
    'Clovers',
    'Diamonds',
    'Horseshoes'
]
```

Arrays are essentially objects

- The object below would work exactly the same as the array on the last slide!

```
var luckyCharms = {
    '0': 'Hearts',
    '1': 'Stars',
    '2': 'Moons',
    '3': 'Clovers',
    '4': 'Diamonds',
    '5': 'Horseshoes'
}
```

Arrays are essentially objects

- Or, you could give your properties names!

```
var luckyCharms = {
    'pink'   : 'Hearts',
    'orange': 'Stars',
    'yellow': 'Moons',
    'green'  : 'Clovers',
    'blue'   : 'Diamonds',
    'purple': 'Horseshoes'
}
```

Objects can store many data types

```
var charlie = {
        age: 8,
        name: "Charlie Brown",
        likes: ["baseball", "The red-haired girl"],
        pet: "Snoopy",
        bald: true
}
//Notice our object contains a number, string, array AND boolean!
Objects are powerful and portable.
```

Returning Object Values

```
var charlie = {
        age: 8,
        name: "Charlie Brown",
        likes: ["baseball", "The little red-haired girl"],
        pet: "Snoopy",
        bald: true
};
charlie.pet; //Call as dot notation (method)…
charlie['pet']; //…or in bracket notation
```

Changing Object Values

- Use dot or bracket notation to change objects values
- Change existing properties:
  - `charlie.name = "Chuck";`
- Or add new properties:
  - `charlie.gender = "male";`
- You can also delete properties:
  - `delete charlie.gender;`

Arrays can hold objects, which are sort of like arrays,
which can also hold arrays...

Arrays of Objects

- Arrays can hold objects, too
- You can loop through an array of objects

```
var peanuts = [
    {name: "Charlie", pet: "Snoopy"},
    {name: "Linus", pet: "Blue Blanket"}
];


for (var i = 0; i < peanuts.length; i++) {
    var peanut = peanuts[i];
    console.log(peanut.name + ' has a pet named ' + peanut.pet + '.');
}
```

Objects in Functions

- Pass an object into a function as a parameter

```
var peanut = { name: "Charlie Brown", pet: "Snoopy" };


function describeCharacter(character) {

        console.log(character.name + ' has a pet named ' +
character.pet + '.');

}

describeCharacter(peanut);
```

Methods

- **Methods** are **functions** inside an object
- They affect or return a value for a specific object
- Methods are used with dot notation, rather than the normal way we're used to calling functions

```
object.method(); //Run the function inside the object

document.write("Hello, world!");
```

Adding methods to objects

- Declare method with the object
- Attached using dot notation

```
var charlie = {
        name: "Charlie",
        sayHello: function() {
                document.write("My name is " + charlie.name);
        }
}
charlie.sayHello();
```

“This”

- Inside methods, properties are accessed using the this keyword
- this refers to the "owner" of the property

“This”

```
var charlie = {
        name: "Charlie",
        sayHello: function () {
                document.write("My name is " + this.name + ".");
        }
};
var lucy = {
        name: "Lucy van Pelt",
        sayHello: function () {
                document.write("My name is " + this.name + ".");
        }
};
charlie.sayHello(); // My name is Charlie.
lucy.sayHello(); // My name is Lucy van Pelt.
```

Nested objects

- Finally, objects can have other objects nested inside them
- Let's go back to our contact example:

```
var contact1 = {
    'FirstName': 'Josh',
    'LastName': 'Collinsworth',
    'CellNumber': 5558675309,
    'HomeNumber': 4815162342,
    'Image': 'images/josh.jpg'
}

console.log( contact1.CellNumber ); //5558675309
```

Nested objects

- It might be easier to combine the numbers into their own object...

```
var contact1 = {
    'FirstName': 'Josh',
    'LastName': 'Collinsworth',
    'Numbers': {
        'Cell': 5558675309,
        'Home': 4815162342
    },
    'Image': 'images/josh.jpg'
}

console.log( contact1.Numbers.cell ); //5558675309
```

2. What is the DOM, anyway?

Not just a cool nickname for someone named Dominic

DOM = "Document Object Model"

- Everything in JavaScript is an object (kind of) (basically)
- The HTML of your site is read by the browser, and then turned into a large, complex object: the DOM, or Document Object Model
- The DOM is an API (application programming interface) for HTML documents
- **"Essentially, it connects web pages to scripts or programming languages."** —MDN

DOM = "Document Object Model"

- Usually, the DOM looks exactly like your HTML
- But it might actually be different, slightly or drastically
- For example: the browser might "fix" your code if you left out something important (if you leave a `<tbody>` tag out of a `<table>`, for example, the browser will insert one for you)
- **But more likely than that: JavaScript can interact with and manipulate the DOM by changing, adding and removing elements, or "nodes"**

DOM nodes

- A "node" is a piece of the DOM
- **A node is basically an HTML element**
- For example, every `<p>` tag in the document is a node. So is every `<ul>` and the `<li>` tags within them, and `<a>` elements within those, and so on
- Even the document itself is technically a node.
- Basically, every building block of the page, big & small, is a node.

DOM nodes

- Here's where it gets confusing: **text is also a node!** (Even white space.)
- So an HTML element that looks like this...
- `<p>This is my paragraph.</p>`
- ...is actually **two** nodes.
- The paragraph tag is a node, and the text inside it is **also** a node (a **text node**, to be precise)
- **JavaScript is capable of treating these separately, and may do so unexpectedly.** That's because a tag and its contents are often two nodes.

DOM nodes

- If this seems confusing, for now, just think of the HTML page as the DOM, and every HTML element on the page as a node
- **JavaScript can interact with, add, remove and modify nodes through an "API"**

3. The DOM API

Doubling down on confusing acronyms

APIs explained

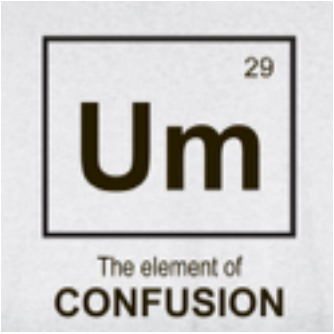
- API = Application Programming Interface
- We can use JavaScript to interact with our HTML pages because the DOM provides us with an API
- The DOM is a web API with methods we can use to modify, add and remove nodes in the DOM and therefore on the page

APIs explained

- Remember: a **method** is basically a command that starts with a dot, like `.log` or `.length`
- All objects have **methods**
- take for example `document.write()`
- `.write()` is the method; the document has a method because the document is an object
- The DOM essentially gives us **methods** to run on our HTML

Um...DOM API nodes...what!?

- If this seems too technical, that's ok
- This is an in-depth explanation of something that you really don't need to understand that well in order to use effectively
- The main idea is: JavaScript has lots of built-in ways to communicate with our HTML



Using JavaScript to interact with the page

- **JavaScript has lots of ways to get specific elements:**

```
document.getElementsByTagName('p')
//retrieves all <p> elements

document.getElementsByClassName('container')
//retrieves all elements with attribute class="container"

document.getElementById('main')
//retrieves the element with attribute id="main"

document.getElementsByName('gender')
//retrieves all elements with attribute name="gender"

document.querySelector('.container')
//retrieves all elements with attribute class="container"
```

Using JavaScript to interact with the page

- **Instead of remembering all of the above, JavaScript gives us a single (somewhat new), handy way to get elements:**
- `querySelector()` and `querySelectorAll()` can be used exactly like CSS selectors or jQuery selectors!

```
document.querySelector(' X ')
//retrieves a single match for X (the first it finds)

document.querySelectorAll(' X ')
//retrieves all matches for X
```

Some examples of Query Selectors

```
document.querySelector('.container')
//Gets the first element with the "container" class

document.querySelectorAll('.container')
//Gets all elements with the "container" class

document.querySelector('#main-nav li')
//Gets the first list item in the element with the
"main-nav" ID

document.querySelectorAll('ul > li')
//Gets all list items that are children of unordered
lists (but no other descendants)
```

Using JavaScript to interact with the page

- Technically, `querySelector()` and `querySelectorAll()` are just slightly slower than more targeted "getters," like `getElementById` or `getElementsByClassName`
- **However, this is rarely important; it's barely a difference**
- Only in extremely large or complex programs should you worry about the distinction

Using JavaScript to interact with the page

- **Usually, you'll assign the retrieved node(s)/element(s) to a variable**
- This makes working with them in JavaScript much easier

```
var allParagraphs = document.getElementsByTagName('p');
//the allParagraphs variable will now contain the contents of every
paragraph element on the page

var containers = document.getElementsByClassName('container');

var main = document.getElementById('main')

var selection = document.getElementsByName('gender')
```

Using JavaScript to interact with the page

- Once you have what you want stored in a variable, you can use JavaScript **methods** on that variable

```
var allParagraphs = document.querySelectorAll('p');
//the allParagraphs variable will now contain the
contents of every paragraph element on the page

console.log(allParagraphs);
[p, p, p, p, p, p, p, p, ...]
```

Using JavaScript to interact with the page

- Since selecting more than one element returns an **array**, looping is usually the best way to affect the selected elements

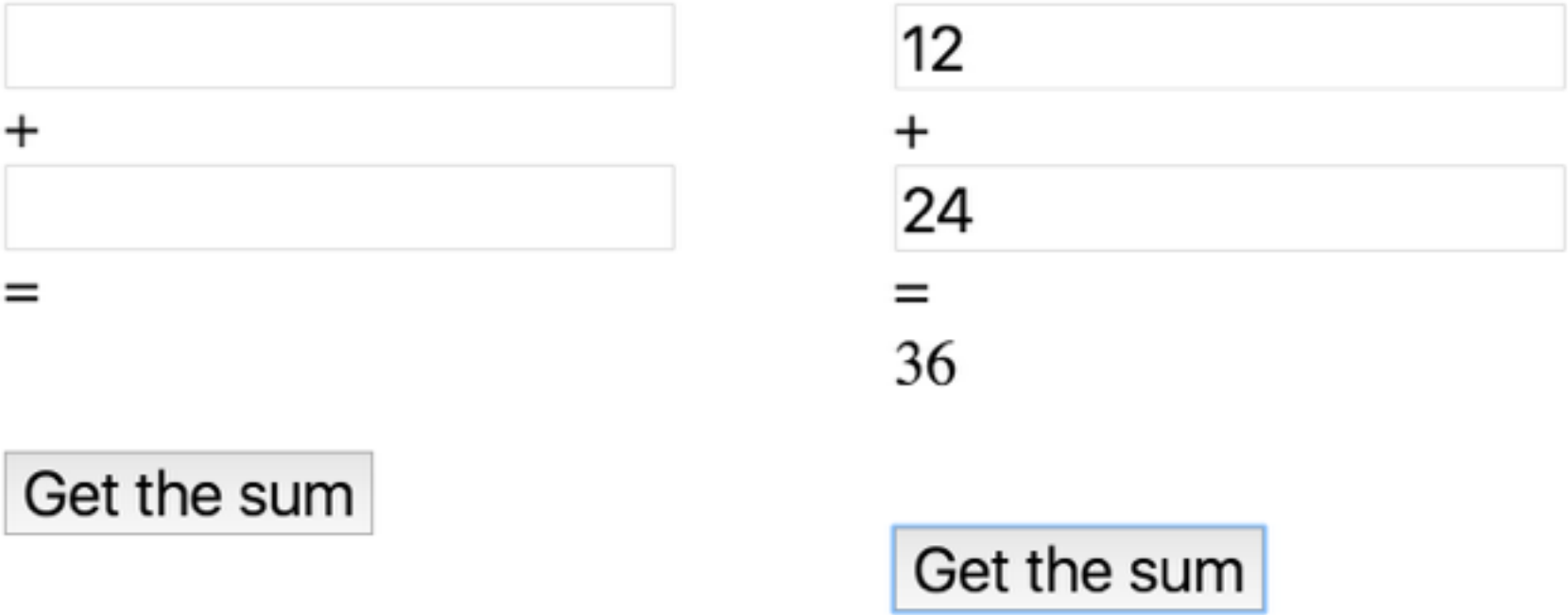
```
var allParagraphs = document.querySelectorAll('p');

for(i=0; i<allParagraphs.length; i++) {
    allParagraphs[i].innerHTML = "Changed!";
}

//Result:
<p>Changed!</p> <p>Changed!</p> <p>Changed!</p> //etc.
```

JavaScript has methods for EVERYTHING

- You won't learn about all that JavaScript can do for a very long time
- Certainly not in this class
- Get in the habit of looking up methods and functions for JavaScript. It can do what you want
- Example searches: "Get value of input JavaScript", "Add HTML to element JavaScript", "change attributes of element JavaScript"
- These will generally take you straight to W3Schools, MDN, or a helpful Stack Overflow post

4. Example One: a Simple Calculator

Let's crunch some numbers!



Let's build a working, simple addition machine!

The HTML: what will we need?

- Think about what elements we'll need for a simple HTML page that takes two numbers and adds them together. What will we need?
- (Hint: there are four main components)

The HTML: what will we need?

- An `input` for the first number
- An `input` for the second number
- A `button` to run the math
- **Optional: a place for the answer to appear (we can also create that place with JavaScript if we want to)**

The JavaScript: what will we need?

- Now, think about the JavaScript commands we'll need to run on the page. Plan ahead. What will we need?
- (Hint: also four main components, but a with a couple of additional optional things and considerations)

The JavaScript: what will we need?

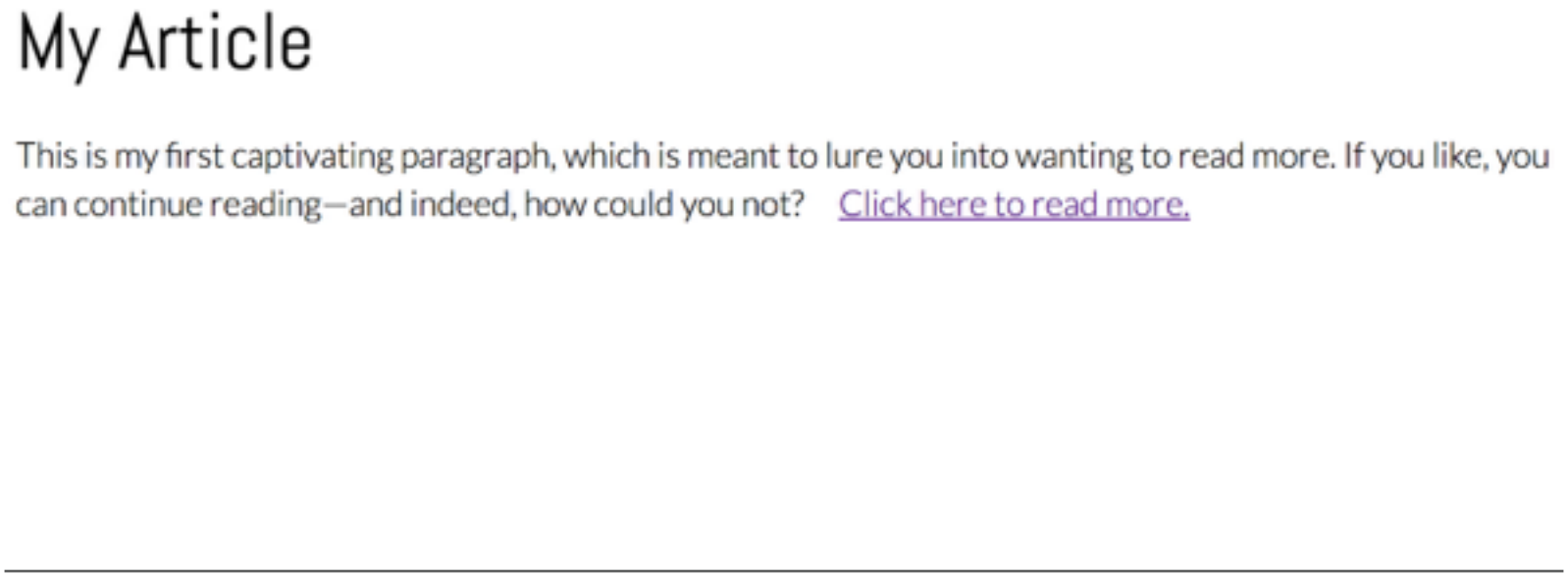
- Retrieve the value from the first input
- Retrieve the value from the second input
- Add the two values together
- Put the result back on the page
- Optional: create a place for the result to go on the page
- **All of this should trigger only when the button is pressed, not before**

Let's build it!

- Create a new HTML document in your text editor, save it, and open it in the browser; or start a new pen on CodePen.io
- Hint: CodePen's console is a little more limited than the browser console, but you can still open the browser console when using CodePen

5. Example Two: a "Read More" Link

Something you can use in the real world



Let's build a working "read more" link!

The HTML: what will we need?

- Think about what elements we'll need for a simple HTML page that takes two numbers and adds them together. What will we need...?

The HTML: what will we need?

- An article of some kind, meaning:
- A header
- At least a couple of paragraphs
- The "read more" link?

The JavaScript: what will we need?

- Now, think about the JavaScript commands we'll need to run on the page. Plan ahead. What will we need...?

The JavaScript: what will we need?

- Hide all paragraphs except the first
- The "read more" link?
- A function that triggers when the link is clicked
- The function should reveal all the hidden paragraphs
- The function should hide the "read more" link

Let's build it!

- Create a new HTML document in your text editor, save it, and open it in the browser; or start a new pen on CodePen.io
- Hint: CodePen's console is a little more limited than the browser console, but you can still open the browser console when using CodePen

6. Progressive Enhancement

An extremely important concept in all development

Progressive Enhancement

- **The practice of making sure your website is accessible and usable to anyone and everyone**
- Rather than break your page for users on old browsers or without certain capabilities, make it work on a basic level, and then **enhance** the page for those who do have greater capability
- **Especially important with JavaScript!**

Why is progressive enhancement important?

- **Don't ever rely on JavaScript just to make your page work.** It should work without JavaScript if it has to (regardless of how ugly or boring that is)
- It's ok if JavaScript adds extra functionality, bells and whistles, etc., but it shouldn't break the page if the script doesn't load or run properly
- There are lots of reasons for this...

Why is progressive enhancement important?

- First, while it's rare, there are some users who don't have JavaScript enabled or are using a device that doesn't support JavaScript. (It's only a tiny percentage, but still)
- Second, the script may not load properly in some cases, like when using a CDN that is down, or if you move your script file and forget to update the link
- Third, a script might break, or there might be bugs unaccounted for
- Fourth, if a script relies on a library (like jQuery), the library may not load properly, which would break your script

The bottom line

- **Build things that work perfectly without JavaScript, then build them to work with JavaScript, too.**
- In our "read more" example, we want to be sure we're hiding our paragraphs with JS and not CSS
- If we hid the paragraphs with CSS and our script broke, the script to show them would never run, and our page would be useless

The bottom line

- By creating our "read more" link and hiding the paragraphs with JavaScript, we ensure that if the page ever loads **without** JavaScript, everything will still work just fine.
- **What about the calculator?** How could we solve the problem of that page loading without JavaScript?

NoScript

- HTML has a `<noscript>` tag
- This tag is an important part of progressive enhancement
- The `<noscript>` tag contains contents that a user will **only** see if JavaScript is disabled or unavailable in their browser
- **IMPORTANT NOTE:** `<noscript>` tags will **not** load when other scripts break; only when JavaScript itself is disabled entirely or not compatible with the current browser

NoScript

```
<script>

    var userName = document.getElementById('nameField').value;

    welcomeMsg = document.getElementById('greeting');

    welcomeMsg.innerHTML = "<h2>Hello, " + userName + "!</h2>";

</script>

<noscript>
    <h2>Hello there!</h2>
</noscript>
```

Questions?

Ask away!

Up Next:

jQuery