Browser JS

- HTML/CSS
    - Browser gets a copy and interprets it
- Browser JS
    - Same thing
    - Gets a *copy*
    - Interprets it

Code runs on THEIR computer, not on the server

- server isn't involved
    - unless the code makes more requests

Rush Tutorial

This is not a Javascript class

- But JS is required for a lot of UI
- We will hit the highlights
    - Get you started
- A lot of nuance and technique not covered

Hello World

```
const message = 'Hello World';
console.log(message);
```

prints "Hello World" to the console

...but how do we run it?

Running in the Console

```
const message = 'Hello World';
console.log(message);
```

You can cut-and-paste this to the browser console

- Good way to test if/how code works
- Not how the page runs code

Code Inline in Element

```
<div onclick="const msg='Hi'; console.log(msg);">Say hello</div>
```

- Special "Event Handler" attributes
- Trigger when certain "events" happen
- Mostly want to avoid this!
    - A real pain to edit
    - Can't reuse
    - Hard to find

Code in Script Element

```
<script>
  const message = 'Hello World';
  console.log(message);
</script>
```

- Generally avoid this too!
    - not as bad as inline in element
    - still can't reuse
    - still JS-in-HTML file
        - Just not JS-in-HTML-attribute
- Executes during render
    - when the lines are reached(!)
    - does not have to be in `<head>`

Code in separate .js file

```
// in a hello.js file
const message = 'Hello World';
console.log(message);
```

```
// in the HTML file
<script src="hello.js"></script>
```

- Reusable, easy to edit
- `src` is a url
    - fully-qualified/absolute path/relative path
- Separate closing tag required (?!)
- Executes during render
    - when the line is reached(!)
    - does not have to be in `<head>`
        - often last element in `<body>`

use strict

JS has some early bad decisions

- old code should continue to work
- new code shouldn't be allowed to still use those bad parts

Putting `"use strict";` at the top of your JS file

- tells browser to require better JS rules

I will often omit this for space/time reasons

- you should still do it in your work

Variables and values

```
const message = 'Hello World';
```

`message` is a variable.

- variables hold **values**
- 'Hello World' is a value
    - a value of type "string"
- JS variables do not have types like some languages (ex: Java)
- JS values DO have types, like Python

Declaring variables

```
const message = 'Hello World';
```

`const` **declares** the variable

- tells the interpreter it exists
- done **once** per variable

Multiple ways to declare a variable

- more on this later

Value types

Different kinds of JS values:

- String
- Number
- Boolean
- null
- undefined
- Array
- Function
- Object

Semicolons

JS is weird

- it has semicolons at the end of statements
- but it works if you omit them

Community divided on this

For this course

- semicolons are REQUIRED
- They offer benefits to scanning code
    - because whitespace isn't signficant to JS

Assignment

Giving a variable a value

- Can be done during declaration

```
const message = 'hello';
```

- ONLY option for `const` variables
- `let` variables can be reassigned

```
let message = 'hello';
message = 'hi'
console.log(message);
```

Prefer `const` over `let`

This is nit-picky

- I require to help teach you subtle impact

Prefer to use `const` whereever you can

- where you don't reassign

`let` is just fine

- but preferring `const` gives more info
- see which variables DO get reassigned

`var` exists

An older way to declare variables

```
var message = 'hello';
```

`var` works, but should be avoided

- You will see it often
- Works in older JS engines
- has some effects that can surprise
    - "hoisting"

String values

- collection of characters
- includes "empty string"
    - (a string with no characters)
- Can be double-quoted: `"Hello World"`
- Can be single-quoted: `'Hello World'`
- A "template literal" is a special type of string
    - still a string
    - created with **backticks**

```
const message = `Hello World`;
```

- quoting method is up to you/team

Template Literals

What makes a template literal special?

- Can be multi-line
- Can **interpolate** values into string

```
const greeting = 'Hello';
const message = `${greeting} World`;
console.log(message); // Hello World
```

Compare to:

```
const message = greeting + " World";
```

Number values

```
const hellYear = 2020;
```

- Tip: JS variables are `camelCased`, not `MixedCase`, `kebab-case`, or `snake_case`
    - Except classes/components are `MixedCase`, and constants are `CONSTANT_CASE`
- Numbers are not quoted
- JS Numbers are "floats"
    - No type for pure integers
- `NaN` is "Not a Number"
    - ironically, `NaN` is considered a Number type
    - represents things like division by a letter

Boolean values

- `true` or `false`
- `||` is "or", and `&&` is "and"
- `!` is "not" - the reverse

```
console.log( true && !false); // true
```

`null` and `undefined`

- These are NOT the same 😿
    - but are very close
- `null` means "set to have no value"
- `undefined` means "never had a value"
    - or "has no matching value"

Rule of thumb: never *set* `undefined`

Advice: use `null` sparingly

Array

```
const cats = [ 'Jorts', 'Maru', 'Grumpy Cat'];
const garbage = [ true, 0, 'test', [ 1, 2 ] ];

console.log( cats[0] ); // Jorts
```

An array is a set of ordered values

- each value can be of any type
    - including nested arrays/objects
- declared with `[ ]`
    - don't use `new Array()` (unneeded)
    - comma separated values
- get elements by a numerical index
    - using `[]`
    - starting at 0

Array methods

Arrays are copied or modified through some methods:

```
const cats = [ 'Jorts', 'Jean' ];
cats.push('Maru');
console.log(cats) // [ Jorts Jean Maru ]
const first = cats.shift();
const last = cats.pop();
console.log(first); // Jorts
console.log(last); // Maru
console.log(cats); // [ Jean ]
```

- `.push()` and `.shift()` let you have a queue
    - FIFO (First In, First Out)
- `.unshift()` and `.shift()` let you have a stack
    - LIFO (Last In, First Out)

Array Length

If you need the length of an array

```
const cats = [ 'Jorts', 'Jean' ];
console.log( cats.length ); // 2

cats.push('Maru');
console.log( cats.length ); // 3
```

- `.length` is NOT a method
- index starts at 0, length starts at 1

```
console.log( cats[ cats.length - 1 ]  ); // Maru
```

Strings as arrays

You can access a string as an array if you need access to specific characters

```
let name = 'Jorts';
console.log( name[2] ); // r
```

But you can't modify the string like you can an array

- For this explicitly convert to an array

```
name.push('!'); // fails, no such method 'push'
```

```
const letters = name.split('');
console.log(letters); // J o r t s
letters.push('!');
name = letters.join('');
console.log(name); // Jorts!
```

See MDN for split/join

Objects

```
const cat = {
  name: 'Jorts',
  age: 3,
  hobbies: {
    activities: ['Twitter', 'trash can'],
  },
};
```

- set of values accessed by a string **key**
- each value can be of any type
    - including nested arrays/objects
- declared with `{}`
    - comma-separated `key: value` pairs
    - key is a string
        - no need to quote simple word keys

Object Notes

- Most important type in JS!
- Basis for most complex data structures
- Often created without a class
- Can create keys using variables

```
const feature = 'color';
const cat = {
  name: 'Jorts',
  [feature]: 'Orange Tabby',
};
console.log(cat.color); // 'Orange Tabby'
```

Accessing Object values

```
const cat = {
  name: 'Jorts',
  age: 3,
  hobbies: {
    activities: ['Twitter', 'trash can'],
  },
};
```

- a value can be accessed using **dot notation**
    - key after a dot after the object name

```
console.log(cat.name); // Jorts
console.log(cat.hobbies.activitites[1]); // trash can
```

- you can also use **index notation**

```
const target = 'name';
console.log(cat[target]); // Jorts
```

Modifying values

Objects and Arrays hold multiple values

- Changing, adding, or removing a values does NOT change the Array or Object
- is considered the SAME Array or Object
    - just different contents
- any other reference to the same Array or Object shows the same change

```
const first = [ 'test' ];
const second = first;
first[0] = 'changed';
console.log(second[0]); // 'changed'
```

Changing an object property

Object properties can be added or deleted

```
const cat = {
  name: 'Jorts',
};
cat.age = 3;
console.log(cat); // { name: 'Jorts', age: 3 }
delete cat.name;
console.log(cat); // { age: 3 }
```

Object Shorthand

Objects can be created using a "shorthand"

```
const name = 'Jorts';
const cat = {
  name,
  age: 3,
};
// same result
const feline = {
  name: name,
  age: 3,
};

console.log( cat.name ); // Jorts
console.log( feline.name ); // Jorts
```

Object Destructuring

You can declare variables based on object properties

- does not modify the object at all

```
const cat = {
  name: 'Jorts',
  age: 3,
  color: 'Orange Tabby',
};

const { name, age } = cat;

console.log(name); // Jorts
console.log(age); // 3
```

This is more common than you'd expect

Functions

Functions are callable code

- they may be passed values
- they may return a value
    - can be array or obj
- a function is a VALUE!
    - can be assigned to variables
    - can be passed as param to functions
- a function is an object
    - (technically lots of types are objects)
- Multiple ways to declare a function

Declaring functions

```
function greet( greeting ) {
  console.log(`${greeting} World`);
}

greet('Hello'); // Hello World

const say = greet;
say('Hi'); // Hi World

const farewell = function() {
  return 'goodbye';
}

farewell(); // (no output)
console.log(farewell()); // goodbye
```

Calling functions

- Without the parens (`()`), you get the function value itself
- With the parens, you get the return value of the function

```
const farewell = function() {
  return 'goodbye';
}

farewell(); // (no output)
console.log( farewell() ); // goodbye

const depart = farewell;
console.log( depart() ); // goodbye
```

Methods

A property of an object can hold a function value

- this is called a **method**

```
const cat = {
  name: 'Jorts',
  meow: function() {
    console.log('meow');
  },
};

console.log( cat.meow ); // (function value)
cat.meow(); // 'meow'
console.log( cat.meow() ); // 'meow' AND undefined - why?
```

Function signature

Function parameters are declared in the **function signature**

```
function greet( greeting, target ) {
  console.log(`${greeting}, ${target}!`);
}
greet('Hello', 'World'); // Hello World!
```

Scopes

Variables are "visible" in a **scope**

- JS is **lexically scoped**
  - a scope can "see" the enclosing scope

```
const name = 'Jorts';

function one() {
  console.log( name );
}

function two() {
  const name = 'Jean';
  console.log( name );
}

one(); // Jorts
two(); // Jean
one(); // Jorts
two(); // Jean
```

Block vs Function scope

A function is a **block**

- so is an if block
- or a for loop block

`const` and `let` variables are **block-scoped**

- `var` variables are **function-scoped**

IIFE

Your code defaults to the **global scope**

- we don't want that

Use an Immediately Invoked Function Expression (IIFE)

```
"use strict";
function() {
  // Everything here is NOT in the global scope
}();
```

I often skip this in examples

- don't skip it in your work

Loops and conditionals

JS has different ways to loop over a collection of items

Can also choose what commands to run based on if something is true

C-style for loop

Common in many languages, including JS

```
for( let count = 1; count < 10; count++ ) {
  console.log(count);
}
// prints 1 2 3 4 5 6 7 8 9
```

Three parts:

- initialization
- check to run before each loop
- step to run at end of each loop

Often used to iterate over an array

For..of loop

Often better than a C-style for loop

```
const cats = ['Jorts', 'Maru', 'Grumpy Cat'];

// C-style
for( let index = 0; index < cats.length; index++) {
  console.log(cats[index]);
}

// for..of
for( let cat of cats ) {
  console.log(cat);
}
```

if/else statement

```
const test = true;
if( test ) {
  console.log('test was true');
} else {
  console.log('test was false');
}
```

- `else` is optional
- `{}` block is optional, but you should always use it
    - REQUIRED for this course
    - prevents confusion about following lines

```
if( test ) {
  console.log('test was true');
}
```

More `if` examples

```
const test = true;
const other = false;

if( test && other ) {
  console.log('Nope');
} else if ( !test || other ) {
  console.log('Still Nope');
} else if ( !test && !other ) {
  console.log('Wow, still No');
} else {
  console.log('Finally);
}
```

Comparison

an `if` statement always forces a **boolean context**

- any non-boolean value will be **coerced** to a boolean
  - This is due to **weak typing**
    - vs **strong typing** like Python or Java
    - distinct from **dynamic vs static typing**
    - Python and JS are dynamically typed
    - Java is statically typed

Strict Comparison

Coercion is usually bad, risks surprises

Avoid coercion by using **strict comparison**

- using `===` or `!==`
- NOT `==` or `!=`

Comparison always returns a boolean value

```
if("1" === 1) {
  console.log('this will not print!');
}

if("1" == 1) {
  console.log('this will print!');
}
```

Explicit Conversion

If you need to compare different types, explicitly convert them.

```
if(Number("1") === 1) {
  console.log('this does print');
}
if( (1).toString() === "1") {
  console.log('this does print');
}
```

Comparing collections

Objects and Arrays will compare **identities**

- not contents

```
const one = [ 1, 2, 3 ];
const two = [ 1, 2, 3 ];
if( one === two ) {
  // they are distinct arrays
  console.log('this will not print');
}
```

How to compare contents depends

- do they have nested contents?
- do they contain functions?

Compare Boolean (truthy/falsy)

You should generally use strict comparison

- An Exception!
- Allow coercion to boolean
    - when the code is easier to read

```
const name = "";

if(name === "" || name === null || name === undefined) {
  console.log('name is required');
}

if(!name) {
  console.log('name is required');
}
```

These are Truthy

Notice: these are TRUTHY

- `[]` (empty array)
- `{}` (empty object)
- `-1` (negative one)

Remember this is FALSY

- `0`

On the Web most input/output starts as string values

What is a callback

A callback is a function passed to another function, so that the receiving function gets control over

- **How many times** (incl 0) to call the callback
- **When** to call the callback
- **What to pass** in the call to the callback

Callbacks are a hugely powerful pattern that allows for code to be written with minimal information, which reduces the complexity, which makes changes easier.

Example callback

```
const students = {
  maru: 87,
  'grumpy cat': 65
};
```

```
const checkGrades = function( students, onStruggle ) {
  for( let name of Object.keys(students) ) {
    if( students[name] < 80 ) {
      onStruggle(name, students[name]);
    }
  }
};

const tellTeacher = function( student, grade ) {
  console.log(`${student} is getting a ${grade}`);
};

checkGrades(students, tellTeacher);
```

Why is that cool

Notice how `checkGrades` doesn't know what you will do with the information!

And yet, `checkGrades` is in control of whether you do it!

Meanwhile, `tellTeacher` doesn't know why it is being called.

In another setup, the exact same `tellTeacher()` could be used to report star students.

In another setup, the exact same `checkGrade()` could be used to email the student a warning.

Callbacks are common

Callbacks are used ALL THE TIME in JS

- *particularly* in frontend
- "just" passing a function as a value

The DOM

The Document Object Model (DOM)

- a hierarchical tree of objects representing the rendered page
- objects have methods to interact with the elements

`window.document`

Because `window` holds all our global variables

`document` is the same value

Finding a node

To interact with an element you need the DOM Node

- `document.getElementById()`
- `document.getElementsByClassName()`
- `document.getElementsByTagName()`

These work

- but we know a flexible way to select element(s)
- `document.querySelector()`
- `document.querySelectorAll()`

read/modify a node

Once we have a node, can read/modify values

- `.value` is the value of input fields
- `.classList` has methods for the classes
    - `.add()`, `.remove()`, `.toggle()`
- `.dataset` lets us set/read special attributes
    - `data-XX` attribute is read as `.dataset.XX`
    - store/read arbitrary related data
- `.disabled` read/set/remove the `disabled` attribute

Writing HTML/Content

In particular

- `.innerText` can read/write the content *text*
- `.innerHTML` can read/write HTML
    - Watch out for security issues!
    - Don't put unsanitized user content in as HTML
    - They could inject their own JS

Events

The user interacting with the page creates **events**

- We can **listen** for those events
- And give **callbacks** to call when they happen

```
const button = document.querySelector('button');
button.addEventListener('click', function() {
  console.log('ow!');
});
```

Common events

You can see events on MDN

Common ones:

- click
- submit (on a form element)
- input (typing/change)
- hover/focus/blur
- invalid (for validation)

Event Propagation

An event fires on an element

- then "bubbles" to the parent
- then its parent, etc

Ancestor elements can have one listener

- that listens to many child elements

Event Queue and loop

When event fires, handler goes in the **event queue**

- Event handler callbacks will NOT run in parallel
  - events will go *into* queue in parallel
- JS is (almost entirely) single-threaded
  - Not the browser, just the browser JS
- Code in queue won't run while other code is
- This cycle is the **event loop**

Little "blocking" code

"blocking" code stops execution

- Little in JS blocks
- instead things are queued and defer to queue
- Exception: `alert()` and similar (`prompt()`)
    - You should NOT use these!
    - Ugly
    - Blocking
    - Makes debugging difficult

Event Object

The event listener calls the event callback

- and passes an event object
- conventionally called `e`
- `e.target` is the element the event actually fired on
- has data about event
    - type
    - related data

Modifying the Event

You can modify the event using the event object

- `.stopPropagation()` prevents the event from bubbling further
- `.preventDefault()` prevents default behavior
    - a link navigating when clicked
    - a form submitting

Example: modifying Events

```
// HTML
<form class="demo-form">
  <button class="button">Please don't click me</button>
</form>
```

```
// JS
const formEl = document.querySelector('.demo-form');
const buttonEl = document.querySelector('.button');

formEl.addEventListener('click', function() {
  console.log("I didn't ask for this!");
});

formEl.addEventListener('submit', function(e) {
  // e.,preventDefault();
});

buttonEl.addEventListener('click', function(e) {
  // e.stopPropagation();
  console.log('ow!');
});
```