We will learn Javascript!

- Essential for modern interactive web pages
- Not a JS course
    - Intro + basic concepts
- Can be overused, poorly used
    - Use right tool for right purpose

Javascript (JS)

- Programming language
    - NOT Java
    - Originally for browser
    - Slight differences per engine
- Runs in browser
    - On user's computer
    - NOT on server
        - "client"/"Browser" vs "server"

JS Capabilities

- Can run on load
- Can react to "events"
    - User actions
    - Network actions
- Can change page HTML
    - HTML structure
    - Text
    - Commonly: classes
- Can cause navigation
- Can submit without navigation!

JS can be messy

- Multiple vendors
- Rapidly evolving web
- "Browser Wars"

Result:

- Many ways to do things
    - They often "work"
- Fewer ways that are "good"
    - But still multiple ways

My advice: 3 year rule

- Just like CSS
- Do not trust articles older than 3 years
    - Too much bad advice
- Lots of articles assume certain libraries
    - They probably don't help you
- If you don't understand it
    - Don't copy/use it

The Browser console

Dev Tools -> Console

- Where `console.log('Hello');` will say "Hello"
- Where error messages will appear
- Erased whenever web page loads/reloads
- Can run commands

Inline JS

- HTML elements can have inline JS
- Just like inline CSS: Don't do this
    - Small exceptions

Do not put JS inline in HTML elements

```
<button onclick="console.log('do not do this')">
  DO NOT
</button>
```

Inline Script Element

- Like CSS
  - Can place inside an element (`<script>`)
- You should not do this
  - Exceptions apply, but not for this course

Do not put your JS inline in a `<script>` element

```
<script>
  console.log("Do not do this, either");
</script>
```

Separate JS File (The Right Way)

- Very similar to CSS
- Load via `src` attribute
    - URL is fully qualified
        - Or absolute path
        - Or relative path
- Still a `script` element
    - But not inline
- Requires closing tag

```
<script src="myfile.js"></script>
```

JS File structure

- Works without this "boilerplate"
    - "works"
- You should always have this boilerplate
    - Until tools do it for us (later)

```
"use strict";
(function() {

  // Your code here!

})();
```

Strict Mode

- Web tries to never "break"
  - Changes can't break older material
  - No one will fix most pages
- Strict Mode enforces newer standards
  - Older content won't trigger Strict Mode
- `"use strict";` as first line sets strict mode
  - Ignored by older engines
- Remember how browser tries to guess?
  - We want strict mode as developers

IIFE - Immediately Invoked Function Expression

- Putting your code in an "IIFE" ("iffy")
    - Prevents accidental creation/overwrite of global values
- Syntax makes sense after functions/scopes
- For now always put your code inside an IIFE
    - "works" without it
- My examples often skip strict/iife
    - Skipped to save space/time
    - You should always have them
        - Until I say otherwise

Your First File

index.html

```
<body>
  ...
  <script src="my-file.js"></script>
</body>
```

my-file.js

```
"use strict";
(function() {
  console.log("Hello World");
})();
```

Where to put `<script>`?

- Many examples show `<script>` in `<head>`
- Causes problems if JS interacts with page content
    - Content doesn't EXIST yet
        - Not yet parsed when parsing is at `<head>`

Two solutions for where to put `<script>`

- Solution One:
  - Put `<script>` as last item in `<body>`
  - Content will exist
  - But can interfere with styling
    - `<script>` is a child of `<body>`
- Solution Two:
  - Put `<script>` in `<head>`
  - Add `defer` attribute
    - Won't execute until content loaded
    - Only works if `src` is used
- You can do either solution for this course

Your first file (take two)

index.html

```html
<head>
  ...
  <script defer src="my-file.js"></script>
</head>
```

my-file.js

```javascript
"use strict";
(function() {
  console.log("Hello World");
})();
```

Variables

```
// Doing more
let message = "Hello";
console.log(message);
```

- The `//` starts a **comment**
  - Goes until end-of-line
  - Purely for humans
- `message` is a **variable**
  - Holds a **value**
- `"Hello"` is a **string value**
  - Collection of **characters**

Declaration

```
let message = "Hello"; // Declare and assign variable
console.log(message);
```

- `let` **declares** our variable
    - Tells JS engine the word is a variable
        - Once per variable
    - Not required unless Strict Mode
        - Anarchy and chaos (and typos)
        - Always "use strict";
    - JS has other ways to declare (later)

Assignment

```
let message = "Hello"; // Declare and assign variable
console.log(message);
```

- `=` is **assignment**
- Assignment has the **variable** hold the **value**
- Assignment does not hold variables!

```
let message = "Hello";
let alternate = message;
message = "Hi";

console.log(alternate); // "Hello" or "Hi"?
```

- Reading a variable gets the value, not the variable

`let` and `const`

- `let`, `const`, and `var` all declare variables
- `var` is old, don't use it
- `const` NOT like other languages
- `const` means "variable does not get reassigned"
    - NOT the same as "constant"
- More later, for now just follow these rules
    - **Use `const` for most variables**
    - **If you reassign the variable, use `let`**
    - **Do not use `var`**
    - Reason: Passively communication

Values have Types

- A value has a **type**
  - What kind of value is it?
  - What can be done with it?
- Example types:
  - Strings (collection of characters)
  - Boolean (`true` or `false`)
  - Numbers (`5`, `8`, `1.5`)
  - More later

Typing

- Languages handle types differently
- In different ways
- **dynamic** vs **static** typing
- **strong** vs **weak** typing
- Two different axis

Dynamic vs Static Typing

- **Dynamic typing**
    - Values have types
    - Variables do not have types
        - Use type of current value
    - Examples: Python, Javascript
- **Static typing**
    - Variables declared with types they can hold
    - Examples: Java, C

```
let message = "Hello";
message = 4; // Dynamic typing (javascript)
```

```
String message = "Hello"; // Static typing (Java)
```

Strong vs Weak Typing

- **Strong typing**
  - Must explicitly change the type
    - Otherwise error
  - Examples: Java, Python
- **Weak typing**
  - Type implicitly converted as needed
    - Known as **coercion**
  - Examples: Javascript, C

```
const num = 4; // Javascript
const message = "Hello-" + num; // "Hello-4
```

```
int num = 4; // Java
String message = "Hello-" + (String) num;
```

Casing

```
let message = "Hello World"; // `let` warns of reassignment
const alternateMessage = message;
message = "Hi";

console.log(alternateMessage); // "Hello World"
```

- `alternateMessage` is multiple words
  - No space
  - No punctuation
  - Later words have first letter capitalized
  - Known as **camelCase**
  - Not **MixedCase** or **snake_case** or **kebab-case**

Semicolons

```
let message = "Hello World";
const alternateMessage = message;
message = "Hi";

console.log(alternateMessage); // "Hello World"
```

- Each instruction (**statement**) ends in `;`
- Everything in JS is **statement** or **block**
    - Statements end in `;`
    - Blocks are in `{}`
        - Usually statements/blocks inside `{}`
- **statements** can be/have **expressions**
    - Expressions evaluate to **values**

JS is very flexible language

- A lot of syntax is *optional*
    - Or has many ways to show it
- Example: Code often works without semicolons
    - Some communities in JS embrace this!
    - **This course requires you use semicolons**
        - Learn it, then you can choose
        - You know a statement ends at end of line
            - Programming is Communication

Some choices are **conventions**

- Example: JS can handle any casing on variables
- **This course requires you use camelCase**
    - Casing is for humans
    - Everyone will use this convention
    - **There are exception**
        - Classes/Components use **MixedCase**
        - constants use **CONSTANT_CASE**

Whitespacing in code for humans

- These two run the same for computer
- Second is better for humans
- Programming is Communication

```
let message="Hello World";
const alternateMessage=message;message="Hi";
console.log(alternateMessage); // "Hello World"
```

Better:

```
let message = "Hello World";
const alternateMessage = message;
message = "Hi";

console.log( alternateMessage ); // "Hello World"
```

Names really matter!

- These two run the same for computer
- Second is better for humans
- Programming is Communication

```
let m = "Hello World";
const aM = m;
m = "Hi";

console.log(aM); // "Hello World"
```

Better:

```
let message = "Hello World";
const alternateMessage = message;
message = "Hi";

console.log( alternateMessage ); // "Hello World"
```

Functions

Functions are a collection of instructions (**statements**)

- Can be **called**
  - Run the instructions
- Can be **passed** values
- Can **return** values
  - Treat as if call is replaced with returned value
- Can be reused
  - Called multiple times
  - Possibly passed different values

Console.log is a function!

- `console.log` is a function
    - The `()` **calls the function**
        - Any values passed are between `()`
        - Comma separated

```
const message = "Hello";
const target = "World";
console.log(message, target);
```

- All functions take **arguments** between `()`
    - Including no arguments (`console.log();`)

Declaring a function

```
function someFunction() {
  const message = "Hello World";
  console.log(message);
}
```

- `someFunction`
    - JS variables (incl functions) are **camelCase**
- Function declaration is a **block**
    - Does not end in `;` (not a **statement**)
- Doesn't output anything!
    - We didn't call it

Calling our function

```
function someFunction() {
  const message = "Hello World";
  console.log(message);
}

someFunction();
someFunction();
```

- Functions called with `()`
- Can be called multiple times
- Function block runs each time

Passing values to a function

```
function greet( name ) {
  const message = "Hello, " + name;
  console.log( message);
}

greet( "Jorts" ); // "Hello, Jorts"
```

- Passed values assigned to variables
    - from function declaration
    - No `let` here

Scopes

- A variable lives in a **scope**
  - Where it can be seen and used
- JS uses **lexical scoping**
  - Nested scopes
  - If variable isn't in current scope
    - Checks enclosing scope(s)
- Same variable name can exist
  - Different scopes = different variables!

Scopes Example

```
const message = "Hello";
const name = "Jorts";
const separator = ", ";

function demo( message ) {
  const name = "Jean";
  console.log( message + separator + name );
}

demo( "Hi" ); // "Hi, Jean"
console.log( message + separator + name ); // "Hello, Jorts"
```

- Each block has a new **scope**
    - Inside existing scope
- Variables for current scope are used
    - Unless not declared in that scope
- Changes only impact variable used

Functions can return a value

```
function makeGreeting( name ) {
  const message = "Hello, " + name;
  return message;
}

// "Hello, Jorts"
console.log( makeGreeting( "Jorts" ) );
```

- **parameters** to function in `()` in declaration
    - comma separated if multiple
- **return** statement gives value to return
- function call replaced with value returned
- **whitespace** is for humans
- function **names** important to understand

More on returning values

- Function block stops executing on return
- Can only return a single value

```javascript
function makeGreeting( name ) {
  const message = "Hello, " + name;
  return message;
  console.log("This does not print");
}

// "Hello, Jorts"
console.log( makeGreeting( "Jorts" ) );
```

Callbacks

- Functions in JS are "first-class citizens"
    - A value like everything else
    - Not true in all languages
- Can assign a function to a variable
- Can pass a function to another function
- Don't use `()` to get value
    - `someFunction` is that function value
    - `someFunction()` CALLS that function
        - evaluates to return of that function call

Callback Example

```
function makeGreat( name ) {
  return name + " The Great";
}

function makeCat( name ) {
  return name + " The Cat";
}

function greet( name, makeTitle ) {
  console.log( makeTitle(name) );
}

greet( "Jorts", makeGreat ); // Jorts The Great
greet( "Jorts", makeCat ); // Jorts The Cat
```

JS Uses Callbacks a lot...why?

- Callbacks let us hand off control
- Example: "When button clicked, call this function"
- We will do this A LOT

DOM - Document Object Model

- JS Data Structure (tree) of the page elements
- Lets us interact with rendered elements
    - Read
    - Edit
    - Delete
    - Respond to **events**
        - Such as "click"
        - Or typing in a form field
        - Or a form "submit"

Finding a DOM Node

- DOM has all elements of page (**document**)
- How to find one of them?
- We *could* use older methods:
    - `document.getElementById()`
    - `document.getElementsByClassName()`
    - `document.getElementsByTagName()`
- But we already know a way to select an element
    - CSS Selectors!

`document.querySelector()`

- Pass it a selector (as a string)
- It returns a value that is a **DOM Node**
- Only returns first matching element
    - Cover multiple shortly

What to do with an element node?

Once you have an element node

- What do you do with it?
- Can read it
    - content text/HTML
    - attribute values
- Can change it
    - content text/HTML
    - attribute values
        - classes
- Can delete it

Example

in HTML:

```html
<button type="button">DO NOT TOUCH</button>
```

In JS:

```js
const buttonEl = document.querySelector("button");

buttonEl.addEventListener("click", function() {
  console.log("Ow! I asked you not to touch!");
});
```

A lot going on there

```
const buttonEl = document.querySelector("button");
```

- Save the DOM Node to a variable

```
buttonEl.addEventListener(...);
```

- Calling some function that's part of the DOM Node
    - More later
- From name: "adding" an "event listener"

```
buttonEl.addEventListener("click", function() {...} );
```

- The name of the **event** we listen for is "click"
- We pass a callback function

Events

- **Events** happen to DOM Nodes
    - like "click", "input", "submit", etc
- We can add **event listeners**
    - Pass **callback functions**
        - Called when event happens
        - Each time event happens

CSS Classes

- Modifying CSS classes is *very* common
- Each DOM node has a `classList`
    - Has some helpful functions on it

Example HTML:

```
<div class="demo active">Some Content</div>
```

Example JS:

```
const demoEl = document.querySelector(".demo");

// demoEl.classList.add("active");
// demoEl.classList.remove("active");
demoEl.classList.toggle("active");
```

Changing class on event

Example HTML:

```html
<button type="button" class="toggle-active">Toggle</button>
<div class="demo active">Some Content</div>
```

Example CSS:

```css
.demo.active {
  background-color: dodgerblue;
}
```

Example JS:

```js
const buttonEl = document.querySelector(".toggle-active");
const demoEl = document.querySelector(".demo");

buttonEl.addEventListener("click", function() {
  demoEl.classList.toggle("active");
});
```

Notice the timing!

- Most of the JS runs on page load
- Callback function runs after event
    - And on each time event happens
- A `console.log()` can tell us when
    - A debugging technique!
    - Remove debugging before submitting

```
const buttonEl = document.querySelector(".toggle-active");
const demoEl = document.querySelector(".demo");

console.log("page loaded");
buttonEl.addEventListener("click", function() {
  console.log("Runs after click");
  demoEl.classList.toggle("active");
});
```

Changing CSS properties

- You CAN change CSS with JS
    - By changing `style` attribute
    - But mostly SHOULD NOT change CSS with JS
        - We don't want to use the `style` attribute
- Instead change CSS classes
    - Have both/all versions of CSS prewritten
    - Like `:hover`/`:focus`
        - CSS exists before it applies

What's with the dots?

- You may have noticed some "dots" in commands:

```
document.querySelector();
console.log();
buttonEl.addEventListener();
demoEl.classList.toggle();
```

To better understand, we need to learn about...

JS Objects

- Objects are a data **type**
    - They are a **value**, like strings and numbers
- Objects are a **collection**
    - Can hold multiple values
        - A value holding values
- Unlike some languages
    - Objects are not all instances of **classes**
        - "classes" different than CSS classes
    - JS Objects are dictionaries or hashmaps

JS Objects are a huge deal

- JS Objects are used so many different ways
- Highly flexible and powerful
- Primary way to collect values
- Can be confusing for those from other languages

JS Object Basics

- Objects hold values by a **key**
  - A string label
  - ANY value type
  - Each key/value is a **property**
- Objects are created by using `{}`
  - **curly braces**
  - Not a block - has semicolon!
  - NOT `new Object()`

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

JS Object Syntax Basics

```
const name = "Jorts";
const cat = {
  name: name,  // Using a variable for value
  age: 3,      // Trailing comma
};
console.log(cat);
```

- Keys before a `:`
    - Keys are strings
    - Do not need to be quoted, UNLESS
        - start with number
        - have spaces or special characters
- Comma (`,`) after value
    - "Trailing" commas allowed

Dot notation

- Properties can be accessed by key

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

- This can be used to modify or add properties

```
cat.age = 4; // changed existing property
cat.color = "Orange Tabby"; // added and set property

console.log(cat);
```

Methods

- A function is a JS value
- An object property can hold ANY JS value
- A property that is a function is called a **method**

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

cat.meow();
```

Looking back at our dot usage

- `document.querySelector();`
  - `document` is an object, root of the DOM "tree"
  - `querySelector` is a method
- `console.log();`
  - `console` is an object, `log` is a method
- `buttonEl.addEventListener();`
  - `buttonEl` is an object, a DOM Node
  - `addEventListener` is a method
- `demoEl.classList.toggle();`
  - `classList` - an object property of a DOM Node
  - `toggle` is a method on the classList object

Nested Objects

- Objects are JS values
- Object properties can hold ANY JS value

Objects properties can hold objects

- `demoEl.classList.toggle()`
- `demoEl` is a DOM Node object
- `classList` is a property of `demoEl`
- `classList` is an object
    - `classList` has properties and methods
        - like `add`, `remove`, and `toggle`

Index notation

- Dot notation works great, except
  - When you are accessing using a variable property name
  - When property name requires quoting

For these cases, we use **index notation** with `[]`

Index Notation with Variable

```
const sleepingCats = {
  Jorts: true, // Properties are normally camelCase
  Jean: false, // Here it is based on data, so not camelCase
};

function wakeUp( name ) {
  // Below would create property "name" :(
  // sleepingCats.name = false;
  sleepingCats[name] = false;
}

wakeUp( "Jorts" );
console.log( sleepingCats );
```

Index Notation with Special Characters

```
const catColorsWeHave = {
  "orange tabby": true,
  "tortoiseshell": true,
};

// Below would look at .gray then fail
// catColorsWeHave.gray tabby = true;

// Below just fails
// catColorsWeHave."gray tabby" = true;

// This works
catColorsWeHave["gray tabby"] = true;
```

Object Mutation

- Most JS values are **immutable**
    - Can replace value, but not change it
    - `num = 4; num = num + 1;` *replaces* 4 with 5
    - Change involves **reassignment** (no `const`!)
- Objects can **mutate**
    - Change a value IN the object
    - Object itself is still same container
    - Object is like a box
        - Changing contents is still same box
- Mutating an object does not change object value
- The object value is the container

Mutation Example

```
const name = "Jorts";
const cat = {
  name: name,
  age: 3,
};

function change( name, cat ) {
  name = "Jean"; // Changes variable in local scope
  cat.age = 5;   // Changes stored value IN object
}

change( name, cat );

console.log( name ); // Jorts
console.log( cat.age ); // 5 ?!
```

Value vs Reference

"Does JS pass by value or by reference?"

- Not an accurate question for JS
- Everything is a value
    - Objects value is the container (a reference)
- Key lessons:
    - Changing a passed variable:
        - Changes value of that variable in scope
    - Changing a value IN a passed value:
        - Changes the value in that container
- Confused? Just remember objects are mutable

Object shorthand highlights differences

```
const name = "Jorts";
const age = 3;
const plaything = "pipe cleaner";

const cat = {
  name: name,
  toy: plaything,
  age: age,
};

const otherCat = {
  name,
  toy: plaything,
  age,
};
```

Creating Object with variable property keys

- Needed rarely
- Can create Object property keys from variables

```
const cat1 = "Jorts";
const cat2 = "Jean";

const sleepingCats = {
  [cat1]: true,
  [cat2]: true,
};

console.log(cat); // { "Jorts": true, "Jean": true }
```

Comparison

Comparing two values

- Gives `true` or `false`
- `===` is "is equal to"
- `!==` is "is not equal to"
- `<`, `>`, `<=`, `>=`
    - Normal math comparisons

```
console.log( "Jorts" === "Jean" );
console.log( 4 < 5 );
console.log( "Jorts" !== "Jean" );
console.log( 8 <= 8 );
console.log( 8 !== 9 );
```

Conditionals (if)

`if` checks if condition is true

```
const num = 8;

console.log(" Thinking... ");

if( num === 8 ) {
  console.log( "Nice!" );
} else {
  console.log( "Not bad" );
}

console.log(" Done ");
```

`if`/`else` are blocks - no semicolons

Conditional Syntax

The `else` is optional

```
if( num === 8 ) {
  console.log("Lucky!");
}
```

You can "chain" many `if`/`else`

```
if( num === 8 ) {
  console.log("Lucky!");
} else if ( num === 9 ) {
  console.log("My favorite");
} else if (num === 1 ) {
  console.log("The loneliest number");
} else {
  console.log("I ran out of space");
}
```

Technically blocks aren't required - USE ANYWAY

```
// Confusing! Always use {} blocks
if( num === 8)
  console.log("Number is 8");
  console.log("This runs no matter what!");
```

Better version:

```
if( num === 8)  {
  console.log("Number is 8");
  console.log("Adding this inside the block!");
}
  // Below is clearly not inside the block
  console.log("This runs no matter what!");
```

Logical operators

- `&&` is logical "and"
- `||` is logical "or"
- `!` is logical "not"
- `( )` group items to ensure order of operations

```
const num = 8;

if( num < 9 && num > 7 ) {
  console.log("Your number is 8"); // or a decimal value
}

if( (num >= 9) || (num <= 7) ) {
  console.log("Your number is NOT 8");
}

if( !true ) {
  console.log("this will never run");
}
```

Identity Comparison

Comparing collections will NOT compare contents

- Instead compares **identity**

```
const one = { name: "Jorts" };
const two = { name: "Jorts" };

if( one === two ) {
  console.log( "This will not print" );
}
if( {} === {} ) {
  console.log( "This also will not print" );
}
```

Comparing contents is complicated

- What if contents include more collections?

More Data Types

We know a bit about:

- String
- Number
- Boolean (`true` and `false`)
- Functions
- Objects

Also have:

- `null` and `undefined`
- Arrays

Nullish values

- A value that means "not a value"
- Can't use 0, because 0 is a value
- Can't use an empty string (`""`), that's a string

JS answers this question TWICE (?!)

- `null` - means "set to not a value"
- `undefined` - means "never had a value"

Together they are **nullish** values

Null

- `null` not used very much
    - Mostly by Java devs learning JS
- Can use to unset a value
    - That's rare

Undefined

- Never explicitly assign a value to undefined
    - That's when you use `null`
- You might *compare* to `undefined`
    - But actually that's rare too
    - Usually check for **falsy** value instead
        - More later
- Variables default to `undefined` value
    - As do object properties

```
let name;
const cat = {};

console.log(name);
console.log(cat.name);
```

Deleting an Object property

- Setting an object property to nullish
    - Does NOT remove the property
- To remove an object property use `delete`

```
const cat = {
  name: "Jorts",
  coat: "Buttered",
};

cat.coat = undefined; // Don't do this!
console.log( cat ); // "coat" property still exists

delete cat.coat;
console.log( cat ); // No more "coat" property
```

Arrays

An **array** is a special kind of Object

- A function is also a special kind of Object!

Normally "object" means non-array, non-function

- But they are all technically objects in JS

Arrays store and retrieve values in **order**

- Use only when you are going access by position
- Position is known as **index**, starts at 0

Creating Arrays

- An array is created with **square brackets**
    - `[]`, comma separated values
- Can hold any JS values
- Values do not have to be same type
    - But usually are the same type
- Trailing commas are okay and common

```
const cats = [
  "Jorts",
  "Jean",
  "Nyancat",
];
```

Accessing Arrays

The individual **elements** (no relation to HTML elements) are accessed using **index notation** and their **position**

```
const cats = [
  "Jorts",
  "Jean",
  "Nyancat",
];

console.log( cats[0] ); // Jorts
console.log( cats[2] ); // Nyancat
```

Arrays are Mutable

- Array values can be changed
    - Doesn't change Array value itself
    - Array is a **collection**, just like Object
        - (Array is also an Object)

Array Methods

Arrays have many useful **methods**

- Some examples (see MDN for details):
- `.slice()` - return new array from parts
- `.splice()` - mutate existing array part
- `.indexOf()` - find matching element index
    - Sign you probably want an object
- `.map()` - return new array based on existing
- `.forEach()` - call callback on each element
- `.filter()` - return new array of filtered elements

Using Arrays

- Common mistake to OVERUSE arrays
    - **You probably want a plain object**
- Use Arrays if and only if:
    - You (almost) always access in same order
    - You are creating a stack data structure
        - use `.unshift()` to add to start of array
        - use `.shift()` to pull from start of array
    - You are creating a queue data structure
        - use `.push()` to add to end of array
        - use `.shift()` to pull from start of array

Comparing Arrays

- Arrays are collections
    - Like Objects (*are* Objects)
- Comparing collections compares **identities**
    - Not contents

```
if( ["Jorts"] === ["Jorts"] ) {
  console.log( "This will not print" );
}
```

- Comparing contents is complicated
    - Contents might have more collections
    - "shallow" or "deep" comparison
        - How deep?

More about Numbers

JS Numbers handle integers AND floating point (decimal)

```
let num = 8;
num += 0.5; // same as num = num + 0.5;

console.log( num ); // 8.5
```

Convert a Number to a String with `.toFixed()`

- use `()` around if not a variable
- or `Math.round()`, `Math.floor()`, etc (see MDN)

```
const num = 8.2345;

console.log( num.toFixed(2) ); // 8.23
console.log( (1234.56).toFixed(1) ); // 1234.6
console.log( Math.floor(1234.56) ); // 1234
```

When is a number not a number?

```
let num = 8;
num = num / "cat";

console.log(num);
```

`NaN` is a special Number value

- Means "Not a Number"
- Doesn't stop program!
    - ...yet

Truthy / Falsy

- Remember **coercion**?
  - Automatic conversion of types
  - Generally bad?
- One type of coercion is good
  - Convert to a Boolean
  - Used in comparison/conditionals
- Conversions to `false` are **falsy** values
  - Conversions to `true` are **truthy** values

Example of Truthy/Falsy benefits

Which do you prefer?

```
if( name !== "" && name !== undefined && name !== null ) {
  console.log("Looks like they entered a name");
}
```

Or:

```
if( name ) {
  console.log("Looks like they entered a name");
}
```

I hope you like the second one better

What is Truthy/Falsy?

- **truthy** values are anything not **falsy**
- **falsy** values will coerce to `false`:
    - `false` (duh)
    - `0` (the number)
    - `""` (empty string)
    - `NaN` (Not a Number)
    - `null`
    - `undefined`
- An empty array (`[]`), empty object (`{}`) are **truthy**
- `"0"`, `"NaN"`, `"null"`, and `"undefined"` are **truthy**

When do you Truthy/Falsy

- When doing a comparison for falsy values
    - Just use truthy/falsy
    - No need for comparison operators at all
        - No `===`, No `!==` to falsy values
- Another use for truthy/falsy: "defaulting"

Short-Circuiting

- `&&` and `||` **short circuit**
- `false && thing` - `thing` doesn't matter
    - won't get evaluated
- `true || thing` - `thing` doesn't matter
    - won't get evaluated
- `&&` and `||` don't return true/false
    - They check arguments for truthy/falsy
    - return the "deciding" argument value

Defaulting a variable

Making sure a variable has a good value

```
let name = someFunction();
if( !name ) {
  name = "Jorts";
}
```

Or

```
let name = someFunction();
name = name || "Jorts"; // name if truthy, else "Jorts"
```

Or

```
let name = someFunction();
name ||= "Jorts"; // same as: name = name || "Jorts";
```

Nullish Coalescing operator (??)

- Sometimes falsy is too wide
  - 0 is falsy
  - "" is falsy
- Sometimes those are okay values
  - You only want to replace **nullish** values
    - `null` and `undefined`

```
let name = someFunction();
name = name ?? "Jorts"; // name can be "", etc
```

Or

```
name ??= "Jorts";  // same as: name = name ?? "Jorts";
```

Loose Comparison

Have you wondered why:

- `=` means assignment
- `===` means comparison? (**strict comparison**)
- What is `==`?

`==` is **loose comparison**

- And you mostly shouldn't use it

Using Loose Comparison - Don't

Loose comparison (`==`)

- Allows **coercion**
    - Which can cause unexpected results

```
if( 1 == "1" ) {
  console.log( "This runs!" );
}
```

Always use strict comparison

- Unless using truthy/falsy
- Which means no comparison at all!

More about strings

Web dev involves a LOT of strings

- Text inside HTML
- CSS class names
- Text values of form fields
    - All form field values are text, even numbers
- Sometimes HTML as a string

JS has a lot of ways to use strings

Quoting Strings

Strings can be quoted multiple ways:

- Double-quoted (`console.log( "Jorts" );`)
- Single-quoted (`console.log( 'Jorts' );`)
- Backtick-quoted (``console.log( `Jorts` );``)
  - Creates a **template literal**
  - Template literals have special abilities

What quoting should I use?

- No common convention
- Some teams prefer one style always
- Some teams use single-quoted for HTML
  - Can easily have double quotes INSIDE string
  - `str = '<nav class="menu">';`
- Some teams use double-quoted for English text
  - Can easily have apostrophes INSIDE string
  - `str = "I'm in favor of Jane's idea";`
- Backticks always solves both
  - But this style still rare

Escaping special characters

`\` before quote character inside string works

- `str = 'I\'m in favor of Jane\'s idea';`
- You can see why people don't like to do this

Template Literals

- Template Literals are special strings
- Can span multiple lines

```
console.log(`

   Hello

`); // can't be done with " or '
```

- Can **interpolate** values into string
    - Dramatic pause...

Template Literals can Interpolate

- Put expression into `${}` inside template literal
  - Replaces with value of expression
  - `expression` is anything that results in a value
    - variable
    - function call
    - calculation

```javascript
const name = "Jorts";

const hard = "I see " + name + ", the Unbuttered is here";
const easy = `I see ${name}, the Unbuttered is here`;

console.log(hard);
console.log(easy);
```

This has all been JS introduction

- Hardly complete
- SO MUCH THOUGH
- A lot of JS syntax
- The idea of the DOM
- `document.querySelector()` to find a DOM Node
    - Node = a rendered element
- Events
- `.addEventListener()` on a node
    - Passed named event type and callback
    - React to named event type ON that element

JS Requirements For This Course (so far)

- Use semicolons
- Prefer `const`, don't use `var`
- Don't use `style` attribute/property
- Use good variable/function names
- Use helpful whitespace
- Load JS as described
- Don't use `style` attribute/property
- Prefer Objects for collections
    - Use Arrays only if:
        - Using order
        - Accessing by position