JS in the browser

Code that runs IN the browser

- Can see the page
- Can change the page
- No page load required
    - faster changes

How to load JS?

How do we get JS running on the page?

- Inline
- In tag
- Separate File

Inline JS

Like CSS, can place as attribute of an element:

```
<span onclick="alert('Hello!')">Cats</span>
```

Demo: Cats

Like CSS, **AVOID INLINE JS**

- pain to edit/maintain/reuse

Also, avoid `alert()`

- blocks
- ugly
- can't modify

In Tag

Like CSS, can place as content of an element

```
<script>
  alert('Heyas');
</script>
```

Like CSS, **AVOID EMBEDDED JS**

- pain to edit/maintain/reuse

Also, avoid `alert()`

- not everyone remembers

As A Separate File

Like CSS, JS can be loaded from a separate file

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

- Same element as in-element approach
    - CSS uses a different element for each way
- `src` instead of `href`
- A distinct closing tag is required

Try it!

JS File:

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

When to load JS

Broswers parse the HTML file from top-to-bottom

- CSS wants to load essential parts early
    - avoid Flash Of Unstyled Content (FOUC)
    - browser pauses to request/parse/apply CSS
- JS often wants to load LATE
    - request/parse/execution of JS can slow page
    - JS interacts with page, page has to be loaded

Most often: put `<script>` tag as LAST part of `<body>`

JS Crash Course

This is not a JS course

- but we will learn a good amount of JS
- taking time beyond this course is good

JS Types of Types

- dynamically typed
    - type goes with value, not variable
- weakly typed
    - interpretation will "coerce" types

Languages have different worlds:

- Python is dynamic-strong
- Java is static-strong
- C is static-weak

Takeaway: No compile time checks, coercion

Not Class-based

JS has objects

- but not (typical) classes
- classes define inheritance
    - but don't limit objects

Values exist outside of objects

- most data is not part of an object
- object types and instances aren't a big deal

Takeaway: Don't rely on `typeof` and `instanceof`

Null and Undefined

JS has **two** kinds of "not defined" values

- `null`
- `undefined`

Generally, `undefined` means:

- never been given a value
- never existed
- not found

`null` means:

- now explicitly set to no value

Booleans

- `true`
- `false`

Not capitalized like Python

- `true && false` (false)
- `true || false` (true)
- `true ? true : false` (true)

Number

No `int` or `float`, just `Number`

- which is actually `float`

Strings

No character type, just strings

3 different quoting syntaxes:

- 'string'
- "string"
    - no difference between them except for which quote you need to escape inside
- template literals

Template Literals

String type using backticks

```
`string`
```

Has some special qualities

- Can span multiple lines
- Can insert variable values using `${variable}`
    - cleaner than multiple strings with `+`

```
`Hello ${name}, nice to meet you`
```

Arrays

An array is an ordered list of items

- items are accessed via their **index**
- Should only be used when order matters!
    - use objects as generic collections

```
const someArray = ['a','b', 'c'];
const otherArray = [];
otherArray.push(1); // otherArray is now [1]
console.log( someArray[1] ); // 'b'
```

Objects - Overview

Objects are huge in JS

- technically any **mutable** value is an object
    - not strings, numbers, undefined, null(mostly)
    - but when we discuss objects, we ignore those other types
- key/value pairs
- act like a `dict`(Python) or `HashMap`(Java)
- values can be any value (properties)
    - incl. functions (methods)
- accessed by key (O(1))

Object - Syntax

- Declared with `{}`

```
const myObj = {};
```

- key/value pairs: `key1: value, key2: value`
- keys are strings, no quotes unless special chars

```
const otherObj = { aProp: 'value', another: 3 };
```

- shorthand uses variable with same name as key

```
const user = 'Preetha';
const someObj = { user }; // { user: 'Preetha' }
```

Object - Accessing

```
const otherObj = { aProp: 'value', another: 3 };
```

- values can be accessed with a `.` and the key
    - "dot notation"

```
console.log( otherObj.another ); // 3
```

- values can be accessed with key name inside `[]`
    - "index notation"
    - key string must be variable or quoted

```
const someVar = 'aProp';
console.log( otherObj[someVar] ); // 'value'
```

Functions - Overview

Functions are the key workhorse of JS

- "First class citizens"
    - treated like any other value
    - assigned, passed, returned
- With `()`, function is called
    - otherwise it is the value

Functions - Declaring

- Declared with `function` or using `fat-arrow`
    - Fat arrow (`=>`) is special, see readings
- list any passed arguments you expect
- No overrides based on "signature"
- No types on parameters

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

const greet = example; // copying
greet('Hello'); // Hello World!
example('Hello', 'random'); // Hello World!
```

Declaring Variables

4 ways to declare variables:

- `var` - don't use, outdated
- `let` - use when variable gets reassigned
- `const` - NOT CONSTANT, prefer to use
- implicit (functions and their params)

var declarations

`var` should only be used in code for old browsers

- common in examples
- declares variable within the same **function**
- allows reassignment
- "hoists"
    - act as if declaration (but not init) was at top

```
var one = 'a word';
var two; // two is undefined, not null
two = [ 5,6,7 ];
```

let declarations

`let` should be used if a variable will be reassigned

- declares variable within the same **block**
- allows reassignment
- does not "hoist"

```
let one = 'a word';
let two; // two is undefined, not null
two = [ 5,6,7 ];
```

const declaration

`const` should be preferred declaration

- ...b/c it makes a need for `let` stand out
- declares variable within the same **block**
- does NOT allow reassignment
    - DOES allow **mutation**
        - changing array or object values
- does not "hoist"

```
const one = 'a word';
const two = [ 5,6,7 ];
two[1] = 8; // allowed
one = 'other'; // ERROR!
```

implicit declaration

- a `function` keyword function *statements* declares a variable
    - not fat arrow
    - like a `var` (hoists, etc)
- function parameters are declared as variables within that function

```
function demo( something, more ) {
  // `demo`, `something`, and `more` are all variables
}
```

Functions as expressions

Functions can be declared as *expressions*

- Does not save name as external variable
- Does not hoist (obviously)
- when this done:
    - passed to function, OR
    - assigned to variable, OR
    - executed immediately

```
const demo = function fromExpr() {
  // fromExpr is a variable in this scope
  console.log('test');
};
// `fromExpr` is not a variable in this scope
```

Conditionals

```
if (condition) {
  // do something
} else if (otherCondition) {
  // do something else
} else {
  // do something else
}
```

Coerces condition to a boolean context

- more on that when we get to **truthy/falsy**

Can omit `{}`, but DON'T

- Creates confusion for maintenance

For loops (C-style)

has `initialization`;`condition`;`post-iteration`

```
for ( let i = 0; i < 10; i++ ) {
  console.log(i);
}
```

Only use if you are actually using the index value(s)

For...of loops

Iterate over the elements of an iterable (list)

```
const demo = [ 'a', 'b', 'c' ];
for( let item of demo ) {
  console.log(item);
}
```

Semicolons

Topic of debate

- JS uses semicolons
    - but will "guess" if not present
- different sub-communities use or don't use

For this course: **Must use semicolons!**

- Aids in skimming code
- Python you know if a line is ended, JS you don't

Statements (if/for/function statements) do not end in semicolons

Coercion

`coercion` is translating between data types

```
console.log( 1 + 5); // 6
console.log( "4"/2 ) // 2
console.log( "test" + 1 ) // "test1"
console.log( "5" + 1 ) // "51"
```

Usually a BAD thing

- except for truthy/falsy (almost there)
- avoid by explicit conversion
- avoid by strict comparison (next)

Comparison

- `=` is assignment
- `==` is loose comparison (coerces)
- `===` is strict comparison (no coercion)

```
1 == '1'; // true
1 === '1'; // false

1 != '1'; // false
1 !== '1'; // true
```

Truthy/Falsy - Overview

Truthy/Falsy values are values coerced to boolean

Useful because they can reduce "visual noise"

- draw eye to variable, not the `===`

```
const user = '';
if( !user ) {
  console.log('You must enter the user name');
}

if( user !== null && user !== undefined && user.length > 0 ) {
  console.log('That was way too much garbage');
}
```

Truthy/Falsy rules

- anything not falsy is truthy
- falsy values
    - `null`
    - `undefined`
    - `''` (empty string)
    - `0` (the number 0)
    - `NaN` ("Not a Number, result of invalid math)
    - `false` (obviously)

```
if ( "test" ) { } // true
if ( 0 ) {} // false
if ( "0" ) {} // true
if ( undefined ) {} // false
```

Nullish Coalescing Operator

- sometimes `0` is valid input
- "nullish coalescing operator" (`??`)
    - an OR where falsy is only null/undefined/false

```
if ( 0 ) {} // false
if ( 0 ?? false ) {} // true
```

Scope

"Lexical Scoping" - current block, fallback to enclosing

```
const cat = 'Maru';
const activity = 'sleeping';
console.log(cat, activity); // Maru, sleeping

function outer() {
  const activity = 'eating';

  function inner() {
    const cat = 'Grumpy';
    console.log(cat, activity);
  }

  console.log(cat, activity); // Maru, eating
  inner(); // Grumpy, eating
  console.log(cat, activity); // Maru, eating
}
outer();
console.log(cat, activity); // Maru, sleeping
```

Short Circuiting

A common pattern for creating default values

- OR (`||`), AND (`&&`), and nullish (`??`) return the first value that determines truthiness
- assign the result

```
function thank(name) {
   name = name || 'fellow student';
   console.log(`Thanks, ${name}!`);
}
thank('Sunil'); // "Thanks, Sunil!"
thank(); // "Thanks, fellow student!"
```

Rest operator

The "rest operator" (`...`) can gather additional function params into an array

- the "spread operator" is also `...`, so watch the context

```
function annoy ( start, ...extra ) {
  console.log(start);
  for( let item of extra ) {
    console.log(`and also ${item}`);
  }
}

annoy("meow");
annoy("woof", "woof", "drool", "poop");
```

Spread Operator

"spreads"

- elements of array (iterable)
- key/value pairs of object

```
const one = [ 1, 2, 3 ];
const two = [ 0, ...one, 4 ]; // 0, 1, 2, 3, 4

const myObj = { name: 'Maru' };
const cats = { ...myObj, other: 'Grumpy' };
  // { name: 'Maru', other: 'Grumpy' }
```

Strict Mode

JS is amazingly backwardly compatible

- but there is still some bad ideas in the language
- avoid the worst by being in "strict mode"

`"use strict";` (the string) as first line of file or function

- tells engine to be more strict
- ignored as thrown away string by old engines

I won't always show this in examples (no space)

- You should do it though

Immediately Invoked Function Expression (IIFE)

When outside a function

- executing in global scope
- hoisted values (`var` and `function` statements)
    - are placed in global scope

This is bad, it can overwrite built-in JS functionality

Wrap all code in a function expression and call it:

```
(function IIFE() {
  // Your code here
  function demo() {
    console.log('exists');
  }
  demo();
})();
```

Callbacks

Functions are "first class citizens" in JS

- Can pass a function to another function
- Called a "callback"
- function decides when and how to call callback
- very powerful pattern

JS Event handlers are all callbacks

- functions passed to be called at the right time

Destructuring

"Destructuring" lets you declare a variable and initialize to a value from an array/object

```
const [ one, two ] = [ 1, 2, 3 ]; // positional
console.log( two ); // 2

const { name } = { name: 'Maru', age: '8 }; // by name
console.log( name ); // 'Maru'
```

Can fake "named parameters"

```
function failure({ name, grade, strict }) {
  if(strict && grade < 80) {
    console.log(`${name} is failing!`);
  } else if (grade < 70) {
    console.log(`${name} is failing!`);
  }
}
failure({ name: 'Maru', grade: 86, strict: false });
// compare to: failure('Maru', 86, false);
```

Document Object Model (DOM)

The DOM is a hierarchical tree of nodes

- the tree of elements on the rendered page
- can read/change/add/remove these nodes

Finding a DOM Node

```
<p id="demo" class="example">Hi</p>
```

```
document.getElementById('demo'); // node
document.getElementsByClassName('example'); // HTMLCollection
document.getElementsByTagName('p'); // HTMLCollection
```

Easier: use CSS Selector syntax

```
document.querySelector('#demo');
document.querySelector('.example');
document.querySelectorAll('.example'); // NodeList
document.querySelectorAll('p'); // NodeList
```

HTMLCollection/NodeList are "array-like" collections

Reading from a DOM Node

- `.value` reads the `value` attribute
- `.innerText` reads the text content
    - including descendants
- `.classList` is an object with methods
    - `.classList.contains()` returns true/false for named class
- `.dataset` is an object with properties that map to `data-XXX` attributes
    - `<span data-test="1" data-other-test="2">`
        - gives `{ test: "1", otherTest: "2" }`

Writing to a DOM node

- Write to properties like `value`
- Add/change values in `dataset`
- `.add()`, `.remove()`, and `.toggle()` on `.classList`
- `.disabled`, `.readonly` to toggle that attribute
- `.innerText` to replace contents
- `.innerHTML` to replace contents
    - Can add HTML nodes!
    - Be cautious of user input!

JS Events

"Events" are when the browser has some activity it might respond to

- clicks
- typing
- scrolling
- moving the mouse
- submitting a form

Events all happen to an **element**

Event Listeners and Handlers

Listener == Handler

You tell JS to respond to a specific event on an element

- give a callback function to call
- call `.addEventListener()` on that element
    - pass event type and callback

JS Event Loop only lets one handler run at a time

- Triggered event handlers wait until they can run

Event Handlers

```
<button type="button">Click Me!</button>
```

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

Event Object

The event handler is passed an event object

- ignored if you don't declare it in your callback
- commonly abbreviated to `e`
- event object has data about the event itself
    - `e.target` is the node the event happened to
    - ...such as the button that was clicked

```
<button type="button">Click Me!</button>
```

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

Event Propagation/Bubbling

Event Propagation (Event Bubbling)

- After handlers of event on element are done
- Same event triggers on parent of element
    - and up to root of DOM Node
- Can halt this with `e.stopPropagation();`

Useful when many nodes all react to same event

- add/remove handlers tedious and error-prone
- just put one handler on ancestor
    - check to see if `e.target` is a node to react to

Default Actions

Some events have default responses

- such as navigating when a link is clicked
- or submitting a form

These happen AFTER your handlers

- Stop default action by calling `e.preventDefault();`