JavaScript

JavaScript

- Interpreted *and* just-in-time compiled (in V8 Engine)
- Java-like syntax, but with functional roots
  - JavaScript syntax was almost functional, like Scheme (i.e. Racket)
- A prototype-based object-oriented language
- Dynamic types (like Python)
  - not *static types* (like Java)
- "Weakly" or "loosely" typed
  - not "strongly" typed (like Python)
- JavaScript is based on an evolving standard
  - ECMAScript: ES6, ...
- Has a *strict mode* and a *non-strict mode* (i.e. "sloppy mode")
  - Vite setup uses strict mode, essentially so does TypeScript

Basics

- Semicolon line endings are optional!
  - but recommended
- "printing" to the console (e.g. of a browser)

```
console.log( ... )
```

- Familiar C/Java style control structure statements, e.g.:

```
if then else
for
while
switch
```

- ternary operator

```
A ? B : C   // equivalent to "if A then B else C"
```

Variables

- Two
  ~~Three~~ ways to declare variables: ~~`var`~~, `let`, `const`
  (`var` has non-block scope "hoisting", generally you don't want that)
- Primitive types:
  - `boolean`, `number`, `string`, `null`, `undefined`
  - *everything** else is an `object`, including arrays
- `undefined` means variable is not defined
- automatic type conversion
- `typeof` statement to return ***primitive type*** as string

often is “object”,
even for things
like an array

* there are some other primitive types like Symbol, ...

Truthy and Falsy

- Automatic type conversion causes some surprising behaviour
- Surprising results of == equality comparisons, e.g.

```
0 == ""  // true!
1 == "1" // true!
```

- Generally, use === for **strict** equality comparisons

```
0 === "" // false
1 === "1" // false
```

https://www.sitepoint.com/javascript-truthy-falsy/

Logical Or and "Nullish Coalescing"

|| is the **logical OR** operator

- often used to assign default value if variable is undefined

```
let v;  // v is undefined
v = v || 456; // v is now 456 since v was undefined
```

- but truthy and falsy behaviour may introduce bugs

```
let v = 0;
v = v || 456; // v is 456 since v was 0 (which is falsy)
```

?? is the **nullish coalescing** operator

- *only* false when `null` or `undefined`

```
let v = 0;
v = v ?? 456; // v is 0 since v wasn’t undefined or null
```

Functions

- Function *declaration*

```
function add1(a, b) { return a + b; }
console.log(add1(1, 2)); // 3
```

- Function ***expression***

```
const add2 = function (a, b) { return a + b; }
console.log(add2(1, 2)); // 3
```

- Function ***expression*** using "arrow notation" / "lambda notation"

```
const add3 = (a, b) => a + b;
console.log(add3(1, 2)); // 3
```

First Class Functions

- Function can be ***assigned*** to a variable

```
function sayHello() { return "Hello, "; }
const saySomething = sayHello;
console.log(saySomething()); // "Hello, "
```

Common for “callback” functions

- Functions can be ***passed*** to other functions

```
function greeting(msg, name) { return msg() + name; }
console.log(greeting(sayHello, "Sam")); // Hello, Sam
```

- Functions can be ***returned*** from other functions

Called "factory pattern" or "factory functions"

```
function makeGreeting() {
  return function (name) { return “Hi ” + name; }
}

const greet = makeGreeting();
console.log(greet("Sam")); // Hi Sam
```

an anonymous function

Closures

- When an inner function references state of outer function

```
function makeRandomGreeting() {
  const sal = Math.random() > 0.5 ? "Hi" : "Hello";
  return function (name) { return sal + " " + name; }
}
const greeting = makeRandomGreeting();
console.log(greeting("Sam")) // ?? Sam
```

- Outer state includes function parameters

```
function makeGreeting(sal) {
  return function (name) { return sal + " " + name; }
}
const greeting1 = makeGreeting("Hello");
console.log(greeting1("Sam")); // Hello Sam
```

Passing Functions to Factory Functions

- Factory function that captures function

```
function makeGreeting(msg) {
  return function (name) { return msg() + name; }
}
function sayHello() { return "Hello, "; }
const greeting2 = makeGreeting(sayHello);
console.log(greeting2("Sam")); // Hello, Sam
```

- Common to use lambda functions in this context

```
const greeting3 = makeGreeting(() => "Howdy! ");
```

an anonymous lambda function

```
console.log(greeting3("Sam")); // Howdy! Sam
```

String Template Literals

- String literal delimited by "backtick" (`) enables:
  - string interpolation
  - multi-line strings
  - tagged templates

we'll talk about these last two later in course

- Example

```
const v = 15.7;
const units = "cm";
```

- Without string interpolation:

```
let msg = "Length is " + v + " " + units + ".";
```

- With string interpolation:

```
let msg =`Length is ${v} ${units}.`
```

- Can use *expressions* in template literal:

```
let msg =`Length is ${(v / 100).toFixed(2) * 100} cm.`
```

formatting method of primitive type

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

JavaScript Objects

- Can be defined using JSON-like* notation (JavaScript Object Notation)

```
const square = {
  colour: "red",
  size: 10,
  draw: function () {
    return `A ${this.size} pixel ${this.colour} square.`;
  }
}
```

- Get property

```
console.log(square.colour); // red
```

- Set property

```
square.colour = "blue";
```

- Call "method" (technically a "function property")

```
console.log(square.draw()); // A 10 pixel blue square.
```

* we don't need to quote property names and some values, plus function property, ...

Prototypal Inheritance

- JavaScript has no formal separation of "classes" and "objects"
- Objects are linked to a special object called the "prototype"
  - all objects have a property called `[[Prototype]]`
- The prototype contains properties and methods for linked objects
- There can be multiple prototypes, forming a "chain"
- Objects can be created using a constructor function and **new** keyword

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

Prototype Chain using Constructor Function

```
// a constructor function
function Shape(colour) {
  this.colour = colour;
  this.draw = function () {
    return `A ${this.colour} shape.`;
  }
}

function Square(colour, size) {
  Shape.call(this, colour);
  this.size = size;
  this.draw = function () {
    return `A ${this.colour} square with size ${this.size}`;
  }
}

const square = new Square("red", 10);
```

"this" refers to object context

call prototype constructor and link to this object

a "shadow" property

Class (like a “template” for creating objects)

- `class` keyword is an abstraction for the prototypical inheritance mechanism

```
class Shape {
  constructor(colour) { this.colour = colour; }
  draw() { return `A ${this.colour} shape.`; }
}

class Square extends Shape {
  constructor(colour, size) {
    super(colour);
    this.size = size;
  }
  draw() {
    return `A ${this.colour} square size ${this.size}`;
  }
}

const square = new Square("red", 10);
```

call prototype constructor and link to this object

a "shadow" property

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_classes

Arrays

- Arrays are an example of an *iterable object*
- Some ways to **declare** an Array:

```
let arr1 = [] // empty array with length 0
let arr2 = Array(5); // empty array with length 5
let arr3 = [1, 2, 3, 4, 5]; // populated array
let arr4 = Array(5).fill(99); // 5 elements, all 99
```

“empty” is not a typo!

- Some ways to **iterate** over an array:

```
for (let i = 0; i < arr3.length; i++) {
  console.log(arr3[i])
}

for (const x of arr3) { console.log(x) }

arr3.forEach((x) => console.log(x));
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections

Array Methods

- foreach
- sort
- reverse
- splice
- indexOf

... many more

Note some mutate the array and some don't, so check the docs!

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Common Functional Array Methods

```
let arr3 = [1, 2, 3, 4, 5];
```

- **map** returns array with transformed elements:

```
const arr4 = arr3.map((x) => x * 10);
// [10, 20, 30, 40, 50]
```

- **find** returns first element that satisfies condition:

```
const a = arr3.find((x) => x % 2 == 0);
// 2
```

- **filter** returns all elements that satisfy condition

```
const arr5 = arr3.filter((x) => x % 2 == 0);
// [2, 4]
```

- **reduce** executes a function that accumulates a single return value

```
const arr6 = arr3.reduce((acc, x) => acc + x, 0);
// 15
```

Destructuring Assignment

- Unpack array elements or object properties into distinct variables
- From Arrays

```
let arr3 = [1, 2, 3, 4, 5];
let [a, b] = arr3; // a = 1, b = 2
```

- From Objects

```
let obj = { "a": 1, "b": 2, "c": 3 };
let { a, b } = obj; // a = 1, b = 2
```

- Can rename destructured variables from objects

```
let obj = { "a": 1, "b": 2, "c": 3 };
let { a: x, b: y } = obj; // x = 1, y = 2
```

means "unpack value for b and store in y"

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Spread Syntax and Rest Syntax

- **Spread** expands an iterable object (i.e. array, string)

```
let arr3 = [1, 2, 3, 4, 5];
let arr4 = [-1, 0, ...arr3, 6, 7];
console.log(arr4); // [-1, 0, 1, 2, 3, 4, 5, 6, 7]
```

- **Rest** condenses multiple elements into single element

```
let arr3 = [1, 2, 3, 4, 5];
let [a, b, ...c] = arr3;
console.log(c); // [3, 4, 5]

const obj = { a: 1, b: 2, c: 3 };
let { a, ...x } = obj;
console.log(x); // {b: 2, c: 3}

let { b, ...y } = obj; // what's y?
```

Create a Prepopulated Array using spread and map

```
let arr5 = [...Array(5)].map((_, i) => i * 10);
```

Spread length 5 empty array

Ignore element value

array index

```
console.log(arr5); // [0, 10, 20, 30, 40]
```

There are many other ways to do this like "from"

Resources for Learning JavaScript

- MDN Introduction to JavaScript
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction
- Strings
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Exercise

1. **Create a Vite Vanilla JavaScript project**
   - npm create vite, then answer prompts and follow instructions
   - Drag folder into VS Code

2. **Check that everything is working**
   - Delete everything In main.js
   - Add a line to console.log “hello” to the console

3. **Experiment with JavaScript concepts**
   - Log some truthy and falsy expressions, including || and ??
   - Create a function that takes a function as an argument
   - Create a simple factory function with a closure
   - Use a string literal
   - Create a simple object (use class keyword)
   - Create an array, try functional methods like map and foreach
   - Use destructuring and spead/rest sytax with arrays and objects