# Section 2: JavaScript Fundamentals – Part 1

## JavaScript

A high-level, object-oriented, multi-paradigm programming language

(Multi-paradigm 🡪 Can be written in different styles eg. imperative and declarative)

## Role of JS

Diagram

Description automatically generated

## Data types

Value is the one that has a type, NOT the variable.

Value can be an object or 1 of 7 primitive data types:

1. Number
2. String
3. Boolean
4. Undefined
5. Null (There’s a legacy bug whereby returns “object”)
6. Symbol (ES2015) 🡪 Value that is unique and cannot be changed
7. BigInt (ES2020)

Diagram

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

## JS operator precedence

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence>

The . in is consider as an operator as well, so it has its own precedence

## Type Conversion vs Type Coercion

|  |  |
| --- | --- |
| **Type Conversion** | **Type Coercion** |
| Manually convert a data type to another | Happens when an operator (arithmetic, logical, comparison, assignment) encounters 2 values of different types  Converts 1 value’s type to match another’s type to ensure the operator’s execution |
| Converts to Number / String / Boolean  (ie. Number(“21”) / String(21) / Boolean (…)) | Eg.  STRING + NUMBER -> NUMBER converted to STRING  STRING – NUMBER -> STRING converted to NUMBER |
| Invalid conversion returns invalid value of that type  (eg. Number(“abc”) -> NaN) | Same as Type Conversion |

## Truthy and Falsy values

|  |  |
| --- | --- |
| **Falsy values** | **Truthy values** |
| Values that become false when converted into Boolean (eg. w/ logical operators or in a logical context) | Values that become true when converted into a Boolean |
| Only 5 falsy values: 0, “”, undefined, null, NaN | Any value that is not falsy |

## Strict vs Loose equality

Strict (=== or !==) 🡪 Both sides have the same data type and value

Loose (== or !=) 🡪 Perform type coercion before comparing their data types and values

## Expression vs Statement

Expression provides a value

Statement indicates an action but does not provide a value directly

## JavaScript Version & History

Text

Description automatically generatedDiagram

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

## Strict Mode in JS (Recommended)

Opting into Strict Mode allows JS to:

1. Throw errors that are otherwise silent
2. Prevent syntax usage that are likely to be defined in future version of ECMAScript

Global 🡪 Need to enter “use strict”; at the 1st line of the .js file

Module 🡪 Strict by default

<https://262.ecma-international.org/6.0/#sec-strict-mode-code>

Text

Description automatically generated

Use Strict Directive 🡪 “use strict”;

## Creating Functions in JS

|  |  |
| --- | --- |
| **Function declaration** | function concat1(str1, str2) {  return `${str1}${str2}`;  }   * Hoisted to the top of the .js file   + Function can be used before the declaration |
| **Function expression** | const concat2 = function (str1, str2) {  return `${str1}${str2}`;  };   * Storing function into a variable since **a function is a type of value** |
| **Arrow function** | const concat3 = (str1, str2) => `${str1}${str2}`;   * Short form of function expression * Cannot use keyword |

## Accessing an Object’s member

|  |  |
| --- | --- |
| **Dot notation** | Party.tanker  Use when the member’s name is known at compile time |
| **Bracket notation** | Party[“tanker]  Use when the member’s name is only known at run time |

## Arrays

A special type of Object in JS, its methods like are just an Object’s method

## Console

An object that provides access to the browser’s debugging console (Eg. Chrome’s F12)

|  |  |
| --- | --- |
| **Notable methods** | |
| console.assert(ASSERTION, obj1 [, obj2, obj3, …]);  console.assert(ASSERTION, msg [, subst1, subst2, …])); | Output a message or a few objects when the given assertion is violated |
| console.warn(obj1 [, obj2, obj3, …])  console.warn(msg [, subst1, subst2, …]) | Output a warning message to the console  1st 🡪 Outputting 1 objects  2nd 🡪 Outputting a message and objects that are used to replace sub-strings (eg. %n or ${…}) in the message |
| console.error(obj1 [, obj2, obj3, …])  console.error(msg [, subst1, subst2, …]) | Output an error message to the console  1st and 2nd same as console.warn(…) |
| console.table(ARRAY\_OR\_OBJ) | Output an array or object in tabular format |

# Section 6 – HTML & CSS

## CSS Box Model

Each element in a page can be seen as a rectangular box like so:



# Section 7 – DOM and DOM Manipulation via JS

## Document Object Model (DOM)

A structured representation of HTML document, generated by the browser on HTML load, to allow JS access elements and styles to manipulate them

* Represented as a tree structure

Diagram

Description automatically generated

* Document is a special object that is the entry point to the DOM (eg. in )
* Properties and methods for DOM are provided by the Web API
* Node refer to a node in the DOM

## Selecting & Manipulating elements

|  |  |  |
| --- | --- | --- |
| **Types of selection** | **Examples** | **Modifying elements** |
| Select an element type |  |  |
| Select an element of class |  |
| Select an element with ID |  |
| **Access/Modify an element’s style** | | |
|  | | |

## Classes in elements

The class(es) of an element can be access through :

|  |  |
| --- | --- |
| **Adding class** |  |
| **Removing class** |  |
| **Checking classes** |  |
| **Toggling class** |  |

Note. Adding/Removal of classes can be used to switch the styling of an element

## Adding UI callback in elements

|  |  |
| --- | --- |
| **Mouse click** |  |
| **Keyboard (keydown, keyup, keypress)** |  |

# Section 8 – How JavaScript works

## Properties of JavaScript

|  |  |
| --- | --- |
| **High-level** | JS has abstractions that manage the resource for us, no need to explicitly manage like in C   * Won’t be as fast as C |
| **Garbage-collected** | Has an algo that remove old/unused objects in the memory (Auto resource management) |
| **Just-in-time compilation** | Entire code is compiled and executed immediately in the machine   * Process: Parse into AST, (Compile into machine code, Execute immediately in Call Stack, Optimize machine code)+ * To allow fast startup of execution, compilation is done quickly which produces unoptimized machine code * While executing, the machine code gets optimized and re-compiled to replace the unoptimized version   Compilation   * Compiled into a binary file and executed some time down the road   Interpretation   * Runs through source code, convert, and execute line-by-line * Very slow |
| **Multi-paradigm** | Paradigm 🡪 An approach of structuring code (eg. Procedural, OOP, Functional)  JS allows those 3 while other languages may only allow one |
| **Prototype-based object oriented** | … |
| **First-class functions** | JS treat functions as variables whereby they can be pass around and return |
| **Dynamic** | Dynamically typed   * Types are only known at runtime and type can easily changed through assignment |
| **Single-threaded** | JS runs in 1 single thread |
| **Non-blocking event loop** | JS runs long-running tasks in the “background” and puts them back to the main thread when they are finished |

## JS Runtime (in browser)

A container for everything needed to run JS in a browser

NOTE. There are other JS runtimes such as Node.JS runtime

Graphical user interface, application

Description automatically generated

|  |  |
| --- | --- |
| JS Engine | A program that read and execute JS code  Heap   * A section of memory to store variables and objects of the JS code   Call Stack   * Stores execution contexts to execute the JS code |
| Web APIs | Provide functionalities to the JS Engine, not part of the JS language  Functionalities provided: DOM manipulation, console, timers, … |
| Callback Queue | Stores the callback functions to call after events occurred. |
| Event Loop | Shift callback functions from the Callback Queue into Call Stack to run when its empty. |

## Execution Context

An environment in which a piece of JS code is executed, and it stores all the necessary info for those code to be executed

* Only 1 global execution context (for code not inside any function)
* 1 execution context per function (created for any function called)
* Contains:
  + Variable environment
    - , , declarations
    - Functions
    - object (Not in arrow functions)
  + Scope chain
  + keyword (Not in arrow functions)

**Process when a JS program is running:**

1. Create a global execution context
2. Execute top-level code through the global context
   1. Create an execution context whenever a function is called
   2. Execute the corresponding piece of code through that context
   3. Remove that context from Call Stack when the function ends/returns
3. If there is no execution context and the program has yet to end, wait for callback functions from the Callback Queue
   1. Do the same as (2) for the callback function
4. Only when the program is terminated (eg. alt-f4), the global context is removed

## Scopes and Scope Chain

|  |  |
| --- | --- |
| **Term** | **Description** |
| Lexical scoping | How a program’s variables are organized and accessed is controlled by the placement of functions and blocks in the code |
| Scope | Region in which a certain variable is **declared** (Global, Function, Block) |
| Scope of a variable | Region in which a certain variable can be **accessed** |

**Types of Scope**

Graphical user interface, application, website

Description automatically generated

NOTE. is function scoped, so, the var is considered in justAFunction()’s scope instead of the if block while is in the if block’s scope:

function justAFunction(fruit) {

if (fruit === “apple”) {

var str1 = “Just an apple”;

const str2 = “Just a fruit”;

console.log(str1);

}

}

NOTE. Within a scope chain, there can be duplicates of variable names. The variable lookup will just return the first found variable with that name when looking up from inner to outer:

function justAFunction(fruit) {

const str = “String 1”;

if (true) {

const str = “String 2”;

console.log(str);

}

console.log(str);

}

The inner console.log will return “String 2” while the outer console.log will return “String 1”.

**Scope chaining**

Every scope has access to the declared variables from all its outer scopes. If not found in current scope, it will look up the outer scopes:

* Only lookup from inner to outer scope

## Variable Environment & Hoisting

**Variable Environment**

Stores all the declarations (let, const, var, functions) and arguments given (as an argument object) in an execution context

* Before execution, code is scanned for declarations and for each variable, a new property is created in the Variable Environment
* When a variable is found to be used before declaration, Hoisting will occur

**Hoisting**

Provide access/usage of variables in the code before they are declared

* Purpose: Allow usage of functions before its declaration
  + hoisting is a byproduct because functions are variables as well

Graphical user interface, text, application, chat or text message

Description automatically generated

**Temporal Dead Zone (for a variable)**

Region of code from start of scope to the declaration of a variable

* Purpose: Easier to avoid and catch errors, make variables work

Diagram

Description automatically generated

## this Keyword

A special variable that is created for each execution context (each function), representing the “caller” of the function

* Depends on how/when the function is called

|  |  |
| --- | --- |
| **Situation** | **Value of** |
| Regular function call | (Strict mode) / (Loose mode) |
| Method call from an object | Object that is calling (eg. Player.attack() 🡪 === Player) |
| Arrow function | of the parent execution context |
| Callback function | Same as Regular function call |
| Event listener’s callback | Element/Object that this function is attached to |
| Others (new, call, apply, bind, …) | … |

**NOTE**

Callback functions are called as a regular function call, resulting in its to be by default

* Since such functions are passed as arguments and called directly without a specific caller, it has no context of

## Regular functions vs Arrow functions

|  |  |
| --- | --- |
| **Regular functions** | **Arrow functions** |
| Refer to above section on what value takes | Lexical scoped and (ie. Refer to parent context’s and ) |
| … | … |

## Primitive variables vs Object variables

|  |  |
| --- | --- |
| **Primitives** | **Object** |
| Primitive variable stores an address that points to a value in the Call Stack | Object variable stores an address that points to an address in the Call Stack  That address points to the value in the Heap |
| Immutable  Modifying the value will involve reserving memory for the new value and setting the variable with the address to that value | Mutable (Actual value, not reference)  Modifying the value will change the actual value within the Heap. Affects any variables that is pointing to the same object |
|  | |

# Section 9 – Data Structures, Modern Operators and Strings

## Destructuring Array

Quick way to assign an array’s items to individual variables.

Example:

|  |  |
| --- | --- |
| **Assignments** | **Comments** |
| dish1 🡪 “Pizza” | - |
| dish3 🡪 “Risotto” | *Skipped “Pasta”* |
| drink1 🡪 “Orange Juice”  drink2 🡪 “Water” | Items from a nested array |
| desert 🡪 “Cheese Cake” | Resort to the default value as is not in |

## Destructuring Object

Quick way to assign an object’s properties to individual variables

Example:

|  |  |
| --- | --- |
| **Possible actions** | **Example** |
| Assign individual variables |  |
| Rename variables |  |
| Setting default values |  |
| Destructure when passing into a function |  |
| Modify existing variables with destructured values |  |

## Spread operator

Quick way of writing multiple values separated by commas

* Supported: Iterables (Arrays, Strings, Maps, Sets), Objects (ES2018)
* Used when building those data structures or passing the values into a function individually

|  |  |
| --- | --- |
| **Spreading array** |  |
| **Spreading object** |  |
| **Conjunction** |  |
| **Passing as parameters** |  |
| **Copying** | Shallow copy 🡪 Clone object by copying memory address  Deep copy 🡪 Clone object by copying values to new memory addresses  Spread operator deep copy the top-level data and shallow copy the sub-level (nested) data  obj.id 🡪 “123” | newObj.id 🡪 “456”  obj.desc.item 🡪 “spear” | newObj.desc.item 🡪 “spear” |

## Rest operator

Quick way of gathering elements or properties into an array or object

|  |  |
| --- | --- |
| **Gather elements/properties when destructuring** |  |
| **Gather trailing parameters that are passed into a function** |  |

## Short-circuiting

|  |  |
| --- | --- |
| **OR operator** | **AND operator** |
| Return the left operand if it is truthy. Otherwise, return the right operand | Return the left operand if it is falsy. Otherwise, return the right operand |
| **Example:**  “” || undefined || 0 || “Toma” || null || 123 🡪 "Toma" | **Example:**  true && 'Toma' && undefined && 123 🡪 undefined |
| **Practical usage:** Assigning default value if does not exist  const coordinates = restaurant.coordinates || { x: 1, y: 2, z: 3 }; | **Practical usage:** Checking existence before accessing it  restaurant.generateErr && restaurant.generateErr() |

|  |
| --- |
| **Nullish Coalescing operator (ES2020)** |
| Return the left operand if it is not nullish. Otherwise, return the right operand.  Nullish values 🡪 null, undefined |
| **Example:** |

|  |
| --- |
| **Optional Chaining (ES2020)** |
| Returns the return value from left operand if it is not nullish. Otherwise, return |
| **Example:**  Chaining a value / an object  const x = restaurant.coordinates?.x;  Chaining a method  restaurant.orderDessert?.(“Cake”);  Chaining an array element  const price = restaurant.mainMenu[3]?.price; |
| **Practical usage:** Provide default value when chaining results in |

## Logical assignment operators

Shortcut to operators used for short-circuiting in situations like

|  |  |  |
| --- | --- | --- |
| **Operators** | **Logical assignment operators** | **Examples** |
| OR operator |  |  |
| AND operator |  |  |
| Nullish Coalescing operator |  |  |

## Iterating through an object’s properties

|  |  |
| --- | --- |
| **How to iterate** | **Examples** |
| By properties’ keys |  |
| By properties’ values |  |
| By properties’ entries  (ie. [key, value]) |  |

## Set & Map

|  |  |
| --- | --- |
| **Set** | **Map** |
| Stores only unique values | Stores key-value pairs of any type (primitives/objects)   * When using object as key, need to reuse memory address (retrieving etc) |
| Represented as an array | Represented as an array of [key, value] |
| **Examples:**  const set = new Set([“a”, “b”, “c”, “b”])  set.has(“a”)  set.add(“d”)  set.delete(“a”)  const uniqueSet = […spellSet]; | Difference from Object   * Object’s key can only be string or symbol * Map is an iterable but Object is not * Map’s pairs in insertion order but Object’s entries ordering is not guaranteed |
| **Map Example** | |
|  | |

Graphical user interface

Description automatically generated

## String

|  |  |
| --- | --- |
| **Get character by index** |  |
| **Get length** |  |
| **Find starting index of substring** |  |
| **Get substring by index** | beginIndex not given  🡪Extract from the beginning  beginIndex < 0  🡪 Extract from str.length + beginIndex  beginIndex >= str.length  🡪 ""  endIndex not given || endIndex >= str.length 🡪 Extract till the end  endIndex < beginIndex  🡪 ""  endIndex < 0  🡪 Extract till str.length + endIndex |
| **Check existence of substring** |  |
| **Change casing** |  |
| **Trimming white space and new line** |  |
| **Replacing substring with another substring** |  |
| **Split string by substring** |  |
| **Padding string with substring** |  |
| **Repeat string** |  |

# Section 10 – Functions

## Default parameters

When declaring a function with parameters, we can assign default values for them:

* Parameters with default value can be not provided:
* Parameters with default value cannot be skipped if the function is providing value to a following parameter:

## Passing parameters (Primitive vs Object)

* Primitive values are passed as a copy of the value
* Objects are passed as references (memory addresses)
  + Modifying it will change the original

## First-class function & Higher-order function

|  |  |
| --- | --- |
| **First-class function** | **Higher-order function** |
| A concept where function are simply values in the programming language (eg. JS) | An implementation of a function where it either takes a function as its parameter or returns a function |
| Treated as a type of object in JS that can be:   * Stored in variables * Pass as parameters * Return from other functions * Call methods on the functions | Only possible because of first-class function |

## Function’s methods (Call, Apply, Bind)

In certain situation, we need the in a function to take on a different value (eg. Reusing an object’s function on another object). Functions in JS has 3 methods to explicitly set the keyword and apply parameters partially:

|  |  |
| --- | --- |
| **Call** | FUNCTION\_NAME.call(THIS\_OBJ, …PARAMETERS)  Calls the function with an explicitly given value and parameters |
| **Apply** | FUNCTION\_NAME.call(THIS\_OBJ, PARAMETERS\_ARR)  Calls the function with an explicitly given value and parameters (as an array) |
| **Bind** | FUNCTION\_NAME.bind(THIS\_OBJ, …PARAMETERS)  Returns a function that is created from a function with an explicitly given and parameters if any   * Parameters can be given partially (ie. 2 out of 4 params given) |

## Immediately Invoked Function Expressions (IIFE)

A disposable 1-time function that run only once immediately after declaration:

* Parenthesis around the function is required to let JS treat it as an actual function
* Aim: Hide variables into another scope (ie. that function’s)
  + Not really needed now with and in a (eg. )

## Closures

A mechanism whereby a function always has access to the variable environment of the execution context where the function was created, even after the context is gone

* Allow the function to access the context’s variables and parameters long after it is gone
* Priority over scope chain

**Practical message – Timer**

# Section 11 – Working with Arrays

Graphical user interface, application

Description automatically generated

## Basic Array methods

|  |  |
| --- | --- |
| **.slice()**  **.slice(BEGIN\_INDEX)**  **.slice(BEGIN\_INDEX, END\_INDEX)** | Returns a shallow copy of a portion of an array from start to end (Similar to ) |
| **.splice(BEGIN\_INDEX)**  **.splice(BEGIN\_INDEX, DELETE\_COUNT)**  **.splice(BEGIN\_INDEX, DELETE\_COUNT, REPLACEMENT\_1, …)** | Modify the contents of an array by removing, replacing, or inserting new elements at an index. Returns the portion of the array that was modified.   * DELETE\_COUNT === 0 🡪 Inserting elements starting from BEGIN\_INDEX |
| **.reverse()** | Modify the contests of an array by reversing the order of array elements |
| **.concat(OTHER\_ARR\_1, …)** | Returns an array that is a merge of 2 or more arrays   * Elements in the new array are all shallow copies |
| **.at(INDEX)** | Similar to [] notation of accessing element via index but can use negative index to iterate from right to left |
| **.forEach(element => {…})** | Run a function using each element in the array |

## Map, Filter, and Reduce

Note. Parameters in the callback function is optional

|  |  |
| --- | --- |
| **.map((ELEMENT, INDEX, ARRAY) => {…})** | Returns a new array containing the results of applying a function on all original array elements |
| **.filter((ELEMENT, INDEX, ARRAY) => {…})** | Returns a new array containing the elements that passed a specified test function |
| **.reduce((ACCUMULATED, ELEMENT, INDEX, ARRAY) => {…})** | Reduces all array elements down to 1 value (including object) through a callback function and an initial value |

## Find, FindIndex

|  |  |
| --- | --- |
| **.find((ELEMENT, INDEX, ARRAY) => BOOLEAN)** | Returns the 1st element that satisfies the given callback function   * if cannot be found |
| **.findIndex((ELEMENT, INDEX, ARRAY) => BOOLEAN)** | Returns the index of the 1st element in the array that satisfies the given callback function   * -1 if cannot be found |

## Includes, Some, Every

|  |  |
| --- | --- |
| **.includes(VALUE)** | Return a boolean on whether the array has VALUE as its element |
| **.some((ELEMENT, INDEX, ARR) => BOOLEAN)** | Returns a boolean on whether the array has at least 1 element that satisfies the given function |
| **.every((ELEMENT, INDEX, ARR) => BOOLEAN)** | Returns a boolean on whether all elements in the array satisfies the given function |

## Flat, FlatMap

|  |  |
| --- | --- |
| **.flat(DEPTH = 1)** | Returns a new array that flatten a nested array times  Eg. [1,2,[3,[4,5]]].flat() 🡪 [1,2,[3,4,5]] |
| **.flatMap((ELEMENT, INDEX, ARR) => ANY)** | Returns a new array that map with the given function first before flatten-ing once |

## Sorting

**Default**

Sort the array by treating the elements as strings and sort them in ascending order of strings

**Given**

Given a comparison function, sort the array elements accordingly by placing the elements like so:

[… < 0 …] b [… > 0 …]

Returns >0 🡪 b before a

Returns <0 🡪 a before b

Returns 0 🡪 Order remains

## Ways of creating array

**[] notation**

Eg. [1, 2, 3, 4, 5]

**Array constructor**

Eg.

new Array(7) 🡪 [empty x 7]

new Array(1,2,3,4,5) 🡪 [1,2,3,4,5]

Note. With only 1 parameter, the constructor will return a new array with that specified number of empty elements

**Array.from(OBJ, (ELEMENT, INDEX) => ANY)**

Eg.

Array.from({length: 7}, (curr, index) => index + 1)

Array.from(new Set(1,2,3,3))

Create a shallow copy Array instance from an array-like or iterable object. If a mapping function is passed, each element will be mapped as well

* Array-like object: An object with a length property (non-negative) and indexed properties
  + Eg. {0: “apple”, 1: “pear”, length: 2}
  + Eg. document.querySelectorAll(…) (ie. NodeList)
* Iterable object: Array, String, Map, Set etc

## Copying using .slice(…) vs spread operator […]

Preferably use .slice(…) as it preserves the sparseness of the array (ie. empty is not preserved):

Eg.

const arr = new Array(3);

console.log([...arr]); // [undefined x 3]

console.log(arr.slice()); // [empty x 3]

# Section 12 – Numbers, Dates, Intl and Timers

## Numbers in JS

All numbers are floating point (ie. decimals) and in base 2 instead of base 10

* Precision representation is impossible where certain decimals cannot be represented

|  |  |
| --- | --- |
| **Action** | **Method** |
| Converting String to Number | (*Work because of type coercion)* |
| Parsing String to Number |  |
| Checking if value is NaN |  |
| Checking if value is Number |  |
| Get max number |  |
| Get min number |  |
| Generate random number |  |
| Truncate number to integer |  |
| Round to nearest integer |  |
| Round up to an integer |  |
| Round down to an integer |  |
| Round to a specified number of decimal places |  |
| Numeric separator  (Increase readability of large numbers) |  |

## BigInt

A data type to represent an extremely large number

|  |  |
| --- | --- |
| **Action** | **Methods** |
| Creating a BigInt | 12321335254365545241565464576456n  BigInt(456254352) |
| Arithmetic operators | +, >, ===, ==, /, … |

## Date

An object that represents a date through an internal representation as the number of milliseconds that passed since Jan 1, 1970

|  |  |
| --- | --- |
| **Actions** | **Methods** |
| Creating a Date | new Date();  new Date(STRING);  new Date(YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, …)  new Date(TIMESTAMP); |
| Getters and Setters | date.getFullYear()  date.getMonth()  date.getDate()  date.getDay()  date.getHours()  date.getMinutes()  date.toISOString()  date.getTime() |
| Get elapsed time | Example – Days passed  (d1, d2) => Math.abs(d2 – d1) / (1000 \* 60 \* 60 \* 24) |

## Internationalization API

Provide features to localize datetime, number, and string. Can provide an Option object to customize the localization

Date Time 🡪 new Intl.DateTimeFormat(LOCALE, OPTION).format(DATE)

Locale can be a specified locale or the browser’s locale (navigator.language)

Option includes etc

Number 🡪 new Intl.NumberFormat(LOCALE, OPTION).format(NUMBER)

Locale is similar to formatting Date Time

Option includes the of number (eg. currency) and other more specific fields

## Timer (setTimeout, setInterval)

setTimeout(CALLBACK\_FUNCTION, DELAY, PARAM\_1, PARAM\_2, …)

Schedule a function to be executed once sometime in the future

setInterval(CALLBACK\_FUNCTION, DELAY, PARAM\_1, PARAM\_2, …)

Schedule a function to be executed every time a preset period has passed

# Section 13 – Advanced DOM and Events

## Document Object Model (DOM)

A structured representation of HTML document, generated by the browser on HTML load, to allow JS access the web page’s elements and styles to manipulate them

* Example of a DOM tree

Diagram

Description automatically generated

* Inheritance in DOM’s API & Types of DOM node

Timeline

Description automatically generated

## Selecting, Creating, and Deleting elements

**Selecting**

document.querySelector…(“…”)

Modern way of selecting element(s) and multiple elements is provided through NodeList (not Live)

* By class: “.CLASS\_NAME”
* By element type: “ELEMENT\_TYPE”
* By ID: “#ID”
* By element that has a specific attribute: “ELEMENT\_TYPE[ATTRIBUTE\_NAME]”
  + ELEMENT\_TYPE[ATTRIBUTE\_NAME=ATTRIBUTE\_VALUE] is also possible

document.getElement…(“…”)

Traditional way of selection element(s) but generally faster in large DOM. Multiple elements are provided through a HTMLCollection that is Live

Live 🡪 Auto-update if an element is added or removed

**Creating**

Inserting HTML

ELEMENT.insertAdjacentHTML(POSITION, HTML\_STRING) 🡪 Parses the HTML\_STRING into a HTML element and added to the POSITION of ELEMENT

Creating HTML element and adding it to another

1. document.createElement(TAG\_NAME) 🡪 Only create the object representing the DOM element, not in the page yet
2. Set the properties of the DOM element via methods like .classList, .ineerHTML etc
3. Add to an existing element in the DOM by selecting one

**Deleting**

ELEMENT.remove() 🡪 Removes the element from the DOM tree it belongs to

## Styles, Attributes, and Classes

**Inline Styling**

* Reading a property
  + Property set with :
  + Property that exists in the element by default:
* Setting a CSS property

**Attributes**

* Attributes that exist in the element by default:
* Non-standard attributes in the element – Not advisable
* Data attributes (accessible through a map of “data-\*” to string) – Standard way to add custom attributes

**Classes**

* An element’s classes can be accessed through which is an array of string

## Event handling

Types of events: <https://developer.mozilla.org/en-US/docs/Web/Events>

2 ways of adding and removing event listeners

* ELEMENT.addEventListener(EVENT\_NAME, CALLBACK) / ELEMENT.removeEventListener(EVENT\_NAME, CALLBACK)
  + Can stack multiple callback functions onto 1 event
* EVENT.onEVENT\_NAME = …
  + Can only assign 1 callback function for each event

## Event bubbling and capturing

**Bubbling and capturing**

When an event happens, the event object gets generated at the root of the DOM tree:

Diagram

Description automatically generated

1. **Capturing phase**

Event object gets passed down all the way to the target element, going through the target element’s parent and ancestor elements. Their callbacks maybe executed if it has been flagged to run in capturing phase

1. **Target phase**

Target element handles the event by executing its callback

1. **Bubbling phase**

Event object gets passed back up to the root element through the parent and ancestors’ elements, calling their corresponding callback if they are listening to the event

* : Stop the event object from being passed further upwards

Note. Event object provides and

🡪 Actual element where the event occurs at

Target 🡪 Element where the eventListener is added to

**Event delegation**

Since events are bubbled up in the bubbling phase, we can place an event listener at a common parent of 1 elements that share the same callback. So, only 1 event listener needs to be added

🡪 Use e.target and e.currentTarget to access the actual element and the parent element

## DOM traversal

|  |  |  |
| --- | --- | --- |
|  | **Parent and Ancestors** |  |
| .parentNode  .parentElement  .closest(“…”) // Closest ancestor |
| **Siblings** | **ELEMENT** | **Siblings** |
| .previousSibling  .previousElementSibling  .parentElement.children | .nextSibling  .nextElementSibling  .parentElement.children |
|  | **Children and Descendents** |  |
| .querySelector(“…”)  .querySelectorAll(“…”)  .childNodes // Nodes  .children // Elements  .firstElementChild  .lastElementChild |

## Passing arguments into Event Handlers

FUNCTION\_NAME.bind(…) allow us to pass an object to be the value of in the function. We can use to provide arguments instead of creating a higher-order function just to do the same thing:

Instead of

## Size and Position of a DOM element

.getBoundingClientRect() gets the position and size of an element

## Interaction Observer API

A way to asynchronously detect the changes in the intersection of target elements with a pre-defined root element like the document’s viewport.

* Uses: Lazy loading images as the page is scrolled, Infinite scrolling websites, Playing animation when reaching an element
* Steps:

1. Create an instance of with the appropriate and
   1. In
      1. Root: The element that the target element check intersection against (null for viewport)
      2. Threshold: How much % of the target element intersect with root (Can be 1 threshold or an array of them)
      3. RootMargin: How much offset from the threshold that the intersection can happen (in pixel)
   2. : (entries, observer) => {…} where entries is a list of that describe the intersection that happened at each threshold
2. Declare what target elements to observe with

## Useful DOM lifecycle events

**DOM Content Loaded**

Fired as soon as the HTML document has been parsed as DOM tree without waiting for other dependencies (stylesheet, images etc)

* .JS that are loaded synchronously with <script> will be loaded when this event is fired
  + - Usually placed at end of <body> to allow most of the page to get loaded first and the DOM elements be available for the .JS

**Load**

Fired when the entire page is loaded including its dependences (stylesheet, images etc)

**Before Unload**

Fired when the page is about to be closed

## Script loading (Sync, Async, or Defer)

Graphical user interface, website

Description automatically generated

Graphical user interface

Description automatically generated with medium confidence

# Regular (Body) vs Defer (Head)

Graphical user interface, application

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

# Section 14 – OOP in JavaScript

## Prototypal Inheritance/Delegation

All objects in JS have a prototype property that links to a prototype object which contains properties and methods that are accessible to all objects linked to the prototype

Text

Description automatically generated

* Since a prototype is an object, it has its own prototype as well
* When an inherited method is called, JS goes through the prototype chain starting from the object to find that method:

Diagram

Description automatically generated

## Constructor functions

A function that returns a new object which contains the following:

1. Properties set by this function (Unique value for each instance)
2. Properties and method inherited from this function’s prototype (Same value for all)

How it works when it is executed:

1. A new {} is created
2. Set this function’s “this” to that, allowing this function to set that object’s properties
3. Set that object’s prototype property to this function’s prototype, providing inheritance
4. Return the object

Diagram

Description automatically generated

**“Class” Inheritance**

“Class” refers to a pair of Constructor function and its prototype.

To inherit the properties and methods from another “class”, we must do the following:

1. In our constructor, call that class’s constructor using our object as
   1. Inherit the properties from the constructor of that class

Text

Description automatically generated

1. Set our constructor’s prototype to an object that uses that class’s prototype as its prototype
   1. Inherit the properties and methods from the prototype of that class



1. Correct the constructor field of our constructor’s prototype to our constructor
   1. Since our prototype is an object that uses that class’s prototype, resetting back to the original is necessary



Diagram

Description automatically generated

## ES6 Classes

Syntactic sugar over Constructor functions and Prototypes but with some notable properties:

1. Not hoisted
2. 1st class citizens (Can be passed around)
3. Always runs in strict mode

Text

Description automatically generated

**Getters and Setters**

|  |  |
| --- | --- |
| **Getter** | **Setter** |
|  |  |
| **Notes** | |
| * Can be accessed like a variable   + INSTANCE\_NAME.GETTER\_NAME   + INSTANCE\_NAME.SETTER\_NAME = … * Can implement direct data validation * If we have getter and setter with the same name as an existing property, use the following pattern: | |

**Static methods**

**Encapsulation**

|  |  |
| --- | --- |
| **Convention (Current)** | **Class fields (Future, still pending)** |
| Having before a property/method’s name to indicate that it is private   * Just a convention to notify other developers, technically not private | Provide the following:   1. Public field 2. Private field 3. Public method 4. Private method 5. Static version of (1) to (4) |
| **Example** | **Example** |
|  |  |

**NOTE:**

Public class fields 🡪 Can be accessed outside of this class and be inherited

Private class fields 🡪 Can only be accessed within this class and cannot be inherited

“Directly set” properties 🡪 Same as Public class fields

**“Class” Inheritance**

Functionally identical to how it works in Constructor functions but just with syntactic sugar

* class XXX extends YYY: Set a X object’s prototype to Y’s prototype, inheriting Y’s methods
* class XXX … {constructor(…) {super(…)}}: Set a X object to an object constructed by Y, inheriting the properties set by Y’s constructor

Text

Description automatically generated

**Summary**

Text

Description automatically generated

## Object.create()

An instance of a class will just be the empty object that uses a given object as its, given by Object.create(PROTOTYPE\_OBJ):

* No concept of constructor functions or keyword to create an instance

Text

Description automatically generated

**“Class” Inheritance**

The same empty object returned from Object.create(…) can be used as a prototype that inherited from another prototype where you can add new properties and methods. Instances from this new prototype can be create with the same Object.create(…):

Text

Description automatically generated

# Section 16 – Asynchronous JavaScript (Promises, Async/Await, AJAX)

## Synchronous vs Asynchronous

|  |  |
| --- | --- |
| **Synchronous code** | **Asynchronous code** |
| Code that are executed right away | Code that are deferred into the future and only executed by then |
| Executed line by line where each line waits for the previous to finish | Executed after a task that runs in the background finishes |
| Long-running operations blocks code execution | Non-blocking |
|  | **Note.** Eventlisteningis not async |

## Asynchronous JavaScript And XML (AJAX)

A technique that allows client to communicate with remote servers asynchronously by request data from them dynamically through an API

* Application Programming Interface (API): Components of a software that can be used by another software to communicate with one another
* Nowadays uses JSON instead of XML

**OLD WAY - Methods and Properties**

|  |  |
| --- | --- |
| **New XMLHttpRequest()** | Constructs a XMLHttpRequest object that handles the asynchronous request |
| **REQUEST.open(REQUEST\_TYPE, URL)** | Set the type of Http request you are making and the target URL |
| **REQUEST.send()** | Send out the request  **Note.** Need to .addEventListener(“load”, …) on request to attach a callback when response arrives |
| **REQUEST.responseText** | The response received from the HTTP request is given in String and needs to be JSON.parse(..) to be an object |

**Callback hell**

Since this old method has no means of chaining requests/asynchronous code, we can only make request 1 after another through nesting callbacks

* Resulted in Pyramid of Doom or Callback hell which is hard to maintain and read

## Promise

An object that is used as a placeholder for the future result of an asynchronous operation.

**Promise Lifecycle**

Diagram

Description automatically generated

**Methods**

|  |  |
| --- | --- |
| **PROMISE.then(FULFILL\_CALLBACK [, REJECT\_CALLBACK])** | Returns a Promise that executes one of the given callbacks asynchronously when PROMISE has been fulfilled or rejected |
| **PROMISE.catch(REJECT\_CALLBACK)** | Returns a Promise that executes the REJECT callback asynchronously when PROMISE has been rejected  Equivalent to PROMISE.then(undefined, REJECT\_CALLBACK) |
| **PROMISE.finally(CALLBACK)** | Returns a Promise that executes a callback asynchronously when PROMISE has been settled (ie. Fulfilled or Rejected) |
| **Promise.resolve(VALUE)** | Returns a Promise that resolves immediately with VALUE |
| **Promise.reject(REASON)** | Returns a Promise that rejects immediately with REASON |

**Note.**

* Any error/rejection a Promise encounters are propagated down to the nearest REJECT\_CALLBACK or .catch(…) to handle
* Error can be through with

**Fetch API**

fetch(RESOURCE, [, INIT])

* RESOURCE: A resource to fetch (eg. URL or Request object)
* INIT: An object containing settings (eg. method to use (eg. GET, POST))
* Returns a Promise object that resolves into a Response object that can be fulfilled or rejected

## How Asynchronous JS works

Graphical user interface, application, website

Description automatically generated

When you execute any asynchronous tasks (eg. Fetching resource, Loading image):

1. Execute it outside of the Call Stack, most tasks are executed in the Web APIs’ environment
2. Callbacks can be attached to the tasks via or via Promise’s API
   1. Callbacks are stored in the same environment as the tasks
   2. Callbacks are set to be executed when a specific event is generated from the task
3. Callbacks to be executed are added to the Callback queue or Microtasks queue
   1. Microtasks queue 🡪 Callbacks from Promises
   2. Callback queue 🡪 Any other callbacks
4. Whenever the Call Stack is empty (except the global context), Event Loop will add the 1st callback in 1 of the queues to the Call Stack for execution
   1. Microtasks queue has the priority

Note:

* Possible for Microtasks queue to starve out Callback queue
* Time-sensitive callback (eg. Timers) may not be punctual as there can be other callbacks in front of it in the queue

## Promisifying

Converting/wrapping callback based async behavior (eg. ) to promise based

For example, converting (…) to :

Text

Description automatically generated

## Async/Await

Synthetic sugar over the chaining promises and callbacks that allow us to write async code with the appearance of synchronous code

* Always return a Promise with the async function’s return value as its fulfilled value

**Error handling - Try…Catch**

* Place the code block into try’s block
* Any error implicitly or explicitly thrown by try’s code block are caught and handled by the catch’s block
* If you want to .catch(…) the returned promise of the async function, you will need to rethrow the error in the catch’s block

**Example – WhereAmI function**

Text

Description automatically generated

## Promise Combinators

|  |  |
| --- | --- |
| **Promise.all (ARRAY\_OF\_PROMISES)** | Returns a Promise that resolves to an array of results from an array of input Promises   * Short circuits when a promise is rejected |
| **Promise.race (ARRAY\_OF\_PROMISES)** | Returns a Promise that resolves to the value returned by the 1st promise that settled (ie. fulfilled or rejected)   * Only 1 promise will be settled, and this overall promise will end |
| **Promise.allSettled (ARRAY\_OF\_PROMISES)** | Returns a Promise that resolves to an array of results from an array of input Promises   * Waits until all promises have settled (resolved or rejected) |
| **Promise.any (ARRAY\_OF\_PROMISES)** | Returns a Promise that resolves to the output of the 1st fulfilled Promise from an array of input Promises   * Short circuits when a Promise is fulfilled |

## Top-level Await (ES2022)

A new feature that allows to be used at the top level (ie. No nesting) without being in an function

* Only valid in Modules
* Can block a Module’s execution while awaiting
  + If there are modules depending on the blocked module, they will be blocked too

Text

Description automatically generated

# Section 17 – Modern JavaScript Development

## Development to Production

Diagram

Description automatically generated

1. Development codes are organized into modules that communicate with one another through importing and exporting. External packages can be introduced too.
   1. Normally 1 module ⬄ 1 file (Always for ES modules)
2. Join all modules (including 3rd party packages) into one .JS file
3. This .JS file is transpiled back to <ES5 code and polyfilled with any untranslatable features to be backwards compatible with older browsers
4. The resulting file is the JavaScript bundle that the client will receive when accessing the website

## Modules

Reusable pieces of code that encapsulates implementation details

* Usually, a standalone file for each module

**Benefits**

* **Reusability**, even across projects
* **Organization**, more organized codebase
* **Isolation**, modules are isolated components, lowering code dependency
* **Compossibility**, modules are building blocks that allow an easier building of applications
* **Abstraction**, low-level code can be done in 1 module and imported into another module

**Structure**

Text

Description automatically generated with medium confidence

## Module pattern

A pattern used to implement modules before any module system is popularized.

**Example**

Text

Description automatically generated

* Access to this function is gone after execution due to usage of IFFE
* const and let variables are block scoped, allowing them to be private to the IIFE
* Allow the developer to pick what to return from the function to serve as the Public API
  + Any returned function has access to the variables due to closure

## ES6 Module System

Modules are stored in files, each module in its standalone file

**Difference from regular script**

Graphical user interface, application

Description automatically generated

* Linking and loading ES6 Module is deferred by nature

**Import / Export syntax**

Named import / export

**Usage**: Exporting multiple things from a module

or

* Export can only be done at top-level code (ie. not in any nesting)

Default import / export

Usage: Exporting 1 thing from a module

* Any name can be used when importing

**Importing a Module**

Example (Importing and into ):

When a module is being imported, it will import and execute all its dependencies before executing itself. So, this "import before execute" behavior will recurse till the lowest level modules:

**Diagram

Description automatically generated**

* All these happen as the modules are imported in an asynchronous manner
* What’s imported is a reference to the value/object instead of the actual copy from the module (Live connection)
  + Changing the imported will affect what’s exported

## CommonJS

A module system used by Node.JS before adopting the ES6 modules

* Many packages still use this system
* Example of the import/export syntax

Text

Description automatically generated with medium confidence

## Module bundling

Joining all modules and packages into a large .JS file with code compression and removal of unnecessary code. This gives the benefits of:

1. Compatibility with older browsers that does not support new features
2. Much faster transfer to client through smaller and fewer files

Library examples: Parcel, Webpack

## Transpiling

Convert post ES6 code to backwards compatible code for older browsers that does not support new features

* Some features are impossible to directly write convert into older JS
* Library examples: Babel

## Polyfilling

Provide modern functionality in older JS by re-creating them from the ground-up

* Eg. Providing Array.find() by re-creating .find() and adding it to Array.prototype
* Library examples: core-js, regenerator-runtime

## Tips to clean code

Graphical user interface, text, application, chat or text message

Description automatically generated Graphical user interface, text, application, chat or text message

Description automatically generated

## Declarative and Functional Principles

Graphical user interface, text, application

Description automatically generated Graphical user interface, text, application, chat or text message

Description automatically generated

# Section 18 – Notes from Forkify

## Imports for polyfilling

import 'regenerator-runtime/runtime'; // Polyfill async await

import 'core-js/stable'; // Polyfill remaining stuff

## Getting relative path to assets in a shipping product

import icons from '../img/icons.svg'; // Parcel 1

import icons from 'url:../img/icons.svg'; // Parcel 2

## Remove any children element in parent element

parentEl.innerHTML = '';

## Mapping an object array to an array of HTMLs to insert into another HTML

*<ul class="recipe\_\_ingredient-list">*

*${*

*// Mapping object to array of strings and joining to a large string*

*recipe.ingredients*

*.map(ingre => {*

*return `*

*<li class="recipe\_\_ingredient">*

*<svg class="recipe\_\_icon">*

*<use href="${icons}#icon-check"></use>*

*</svg>*

*<div class="recipe\_\_quantity">${ingre.quantity}</div>*

*<div class="recipe\_\_description">*

*<span class="recipe\_\_unit">${ingre.unit}</span>*

*${ingre.description}*

*</div>*

*</li>*

*`;*

*})*

*.join('\n')*

*}*

*</ul>*

## Comparing whether set of elements are the same by comparing DOMs

Create a virtual DOM, allowing us to compare with current DOM to check for modification

## Use .findIndex(...) and .splice(...) to remove an element

## Use .some(...) and .every(...) to check if elements satisfy a condition

**When UI events involve elements that may not always be present, consider using event delegation**

e.target.closest(...) can be used to find the closest ancestor that satisfy the given query

To optimize performance, consider **only re-rendering the modified elements** when an event happens to minimize workload of DOM

## Verify a value's existence before adding to an object

**It is common for projects to have Helper and Config modules to provide functionalities and variables that will be used across the project**

## History API

Window object provide access to the browser's session history through the "history" object (ie. Window.history), exposing methods and properties that allow navigation of user's history and manipulating the history stack

**Navigation**:

**Manipulate stack**:

## Node and element relationship

Node is the smallest unit in the DOM tree.

* There are different types of Nodes, including Element, Text, Comment, and Document. Each is responsible for different aspect of an HTML document (Refer to the tree in Notes section 13).
* In this case, Element is a Node that represents a HTML element and contains properties and methods unique to that type of HTML element eg. <p> ... </p> (just the <p> tags, not what's inside)
* **Text** -> The text within a HTML element (eg. TEXT in <p> TEXT </p>)
* **Comment** -> Comment in a HTML document
* **Document** -> The overall document, root node of the DOM

## .nodeValue, .textContent

**.nodeValue**: Value of the current node, differing based on type of node

Comment: Comment itself

Document: Null

Element: Null

Text: Text content itself

**.textContent**: Text content of the node and its descendants

Document: Null

Comment / Text: .nodeValue

Element: Concat of every child node's .textContent

## Existence of text node in element

If there is text between an element's enclosing tag, there will be a child text node in the element node

## URI fragment

A string of characters at the end of an URL that refers to a resource on the page. This string comes after a # like so: http://youtube.com/history#today where the fragment is "today"

* Change in the fragment can be detected via the hashchange event.
* Access to the fragment can be done through "window.location.hash"

## MVC architecture in this project

* Model and Views have no knowledge of Controller, only exposing their API for the Controller to manipulate
  + Controller uses Model to obtain data and passed into View for display
* Views provide a method for Controller to subscribe callbacks to UI events happened in those Views
  + Allowing the appropriate action to be taken by the application