# 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

|  |  |
| --- | --- |
| **Method of calling** | **Value of** |
| Simple call | (Strict mode) / (Loose mode) |
| Calling from an object | Object that is calling (eg. Player.attack() 🡪 === Player) |
| Arrow function | of the parent execution context |
| Event listener | DOM element that this function is attached to |
| Others (new, call, apply, bind, …) | … |

## 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

## 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 through a callback function and an initial value |