# 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 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

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

**Global Execution Context:**

refers to the global object ( in JS browser runtime)

**Standard Execution Context:**

* Depends on how/when the function is called

|  |  |
| --- | --- |
| **Method of calling** | **Value of** |
| Simple call |  |
| 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 |
|  | |