www.chuwaamerica.com

2.1 Introduction to JavaScript

- Created by Netscape
- Version History
- ❖ The Original JavaScript ES1 ES2 ES3 (1997-1999)
- ❖ The First Main Revision ES5 (2009)
- ❖ The Second Revision ES6 (2015)
- ❖ The Yearly Additions (2016, 2017, 2018)

- ECMAScript provides the core functionality.
- The Document Object Model (DOM) provides interfaces for interacting with elements on web pages
- The Browser Object Model (BOM) provides the browser API for interacting with the web browser.

www.chuwaamerica.com

2.1.1 JavaScript vs Java

- Java is considered a compiled programming language, while JavaScript is considered an interpreted scripting language
- Java uses static type checking, and JavaScript uses dynamic typing
- Java uses multiple threads to perform tasks in parallel, whereas JavaScripst handles concurrency on one main thread of execution
- Java follows class based inheritance, while in JavaScript, inheritance is prototypal

2.2.1 JavaScript Syntax : Identifiers

Identifiers

- An identifier is the name of a variable, function, parameter, or class.
- An identifier consists of one or more characters in the following format:
- The first character must be a letter (a-z, or A-Z), an underscore(_), or a dollar sign ($).
- The other characters can be letters (a-z, A-Z), numbers (0-9), underscores (_), and dollar signs ($).
- It is a good practice to use camel case for the identifiers, meaning that the first letter is lowercase, and each additional word starts with a capital letter.

2.2.2 JavaScript Syntax : comment

JavaScript supports both single-line and block comments.
A single-line comment starts with two forward-slash characters (//), for example:

```
// this is a single-line comment
```

A block comment starts with a forward slash and asterisk (/*) and ends with the opposite (*/) as follows:

```
/*
 This is a block comment that can
 span multiple lines
*/
```

2.2.3 JavaScript Syntax : Statements and declarations

JavaScript applications consist of statements with an appropriate syntax. A single statement may span multiple lines. Multiple statements may occur on a single line if each statement is separated by a semicolon.

Statements and declarations by category

- Control flow -> if
- Declarations -> let, const
- Functions and classes -> function
- Iterations -> for

2.3.1 JavaScript Variables : introduction

There are 3 ways to declare a JavaScript variable:

- Using var
- Using let
- Using const

```
var variableName = value;
let variableName = value;
const variableName = value;
```

What is the difference between var, let and const ? We will discuss it later

2.3.2 JavaScript Variables :Global and local variables

- In JavaScript, all variables exist within a scope that determines the lifetime of the variables and which part of the code can access them.
- JavaScript mainly has global and function scopes. ES6 introduced a new scope called block scope.
- If you declare a variable in a function, JavaScript adds the variable to the function scope. In case you declare a variable outside of a function, JavaScript adds it to the global scope.

www.chuwaamerica.com

2.3.2.1 JavaScript Variables : Global and local variables examples

```
 function say() {
    var message = "Hi";
    return message;
}
```

We defines a function named say that has a local variable named message.

The message variable is a local variable. In other words, it only exists inside the function. If we try to visit message outside the function say

```
console.log(message); // ReferenceError
```

www.chuwaamerica.com

2.3.3 JavaScript Variables : Variable shadowing

```
// global variable
var message = "Hello";

function say() {
    // local variable
    var message = 'Hi';
    console.log(message);
    // which message?
}

say();// Hi
console.log(message);  ?
```

In this example, we have two variables that share the same name: message. The first message variable is a global variable whereas the second one is the local variable.

Inside the say() function, the global message variable is shadowed. It cannot be accessible inside the say() function but outside of the function. This is called variable shadowing.

2.3.4 JavaScript Variables : Non-strict mode

```
function say() {
    message = 'Hi';
    console.log(message);
    // which message?
}

say();// Hi
console.log(message);  //Hi
```

In this example, we have two variables that share the same name: message. The first message variable is a global variable whereas the second one is the local variable.

Inside the say() function, the global message variable is shadowed. It cannot be accessible inside the say() function but outside of the function. This is called variable shadowing.

www.chuwaamerica.com

2.3.5 JavaScript Variables : strict mode

```
function say() {
    message = ‘Hi’;
    // ReferenceError
    console.log(message);
    // which message?
}

say();
console.log(message);
```

To avoid creating a global variable accidentally inside a function because of omitting the var keyword, you use the strict mode by adding the "use strict"; at the beginning of the JavaScript file (or the function)

2.3.6 JavaScript Variables : hoisting

```
console.log(message);
var message;
```

```
var counter;
console.log(counter);
counter = 100;
```

When executing JavaScript code, the JavaScript engine goes through two phases:

- Parsing
- Execution

In the parsing phase, The JavaScript engine moves all variable declarations to the top of the file if the variables are global, or to the top of a function if the variables are declared in the function.

Hoisting is a mechanism that the JavaScript engine moves all the variable declarations to the top of their scopes, either function or global scopes

2.3.7 JavaScript Variables : Using let keywords

```
var a = 20, b = 10;
{
        let tmp = a;
        a = b;
        b = tmp;
}
console.log(tmp); // ReferenceError
```

From ES6, you can use the let keyword to declare one or more variables. The let keyword is similar to the var keyword.

However, a variable is declared using the let keyword is block-scoped, not function or global-scoped like the var keyword.

2.3.8 JavaScript Variables : More examples about let and var

```
function increase() {
        var counter = 10;
}
// cannot access the counter variable here
```

When you declare a variable inside a function using the var keyword, the scope of the variable is local.
<- For example

```
for (var i = 0; i < 5; i++) {
        console.log("Inside the loop:", i);
}
console.log("Outside the loop:", i);
```

The example displays four numbers from 0 to 4 inside the loop and the number 5 outside the loop.

In this example, the i variable is a global variable. Therefore, it can be accessed from both inside and after the for loop.

2.3.9 JavaScript Variables : More examples about let and var

```
for (let i = 0; i < 5; i++) {
        console.log("Inside the loop:", i);
}
console.log("Outside the loop:", i);
```

In this case, the code shows four numbers from 0 to 4 inside a loop and a reference error:

www.chuwaamerica.com

2.3.10 JavaScript Variables : const keywords

```
const pi= 3.14; pi = 3.141;

// TypeError: `code` is read-only
```

The const keyword works like the let keyword, but the variable that you declare must be initialized immediately with a value, and that value can't be changed afterward.

2.3.11 JavaScript Variables : Constant Objects and Arrays

```
const constArray = [1, 2, 3];
constArray[0] = 5;
constArray.push(6);
console.log(constArray);
```

How about const Object Array ?

- A little misleading
- You can modify and you can't reassign

www.chuwaamerica.com

2.3.12 JavaScript Variables : Summary for var, let and const

- Var is not recommended to use
- Use let and const as much as possible you can

- If you want to define a let and you want to change the value later, use let, otherwise you can use const directly

- For the object and array, you could use const directly if you don’t want to reassign the value the it.

www.chuwaamerica.com

2.4.1 Data Types

JavaScript has six primitive data types:

- null
- undefined
- boolean
- number
- string
- symbol – available only from ES6

- One complex data type called object

www.chuwaamerica.com

2.4.2 Data Types -The undefined type and null

The undefined type is a primitive type that has one value undefined.

By default, when a variable is declared but not initialized, it is assigned the value undefined.

The null type is the second primitive data type that also has only one value: null

```
let counter;
console.log(counter); // undefined
console.log(typeof counter); // undefined
```

```
let obj = null;
console.log(typeof obj); // object
```

www.chuwaamerica.com

2.4.3 Data Types -The The number type

- Integer numbers
- Floating-point numbers
- NaN: which stands for Not a Number. In fact, it means an invalid number

```
let num = 100;
let pi = 3.14;

Typeof num  // number
console.log('a'/2); // NaN;
```

2.4.3.1 Data Types -The The number type continue

Question: what is the result of 0.1 + 0.2 ?

The result is not 0.3 !

The actual result is 0.30000000000000004

We can use the following method to get the accuracy Num.toFix(1)

2.4.4 Data Types -String

In JavaScript, a string is a sequence of zero or more characters. A literal string begins and ends with either a single quote(‘) or a double quote (“).

You can use literals `` to represent string as well

```
let greeting = 'Hi’;
let s = "It's a valid string";
let str = ‘I\’m also a string‘;  // use \ to escape the single quote (‘)

Let greeting = `HI`
let message = `Hello ${name}`;
```

2.4.4.1 String Built-In Methods

trim() returns a new string stripped of whitespace characters from beginning and end of a string:

```
let inputValue = str.trim();
```

To remove whitespace characters from the beginning or from the end of a string only, you use the trimStart() or trimEnd() method.

2.4.4.2 String Built-In Methods

indexOf() returns the index of the first occurrence of substring (substr) in a string (str). And we have the same built-in method in Array

let index = str.indexOf(substr, [, fromIndex]);

The indexOf() always perform a case-sensitive search.

To perform a case-insensitive search for the index of a substring within a string, you can convert both substring and string to lowercase before using the indexOf() method

let index = str.toLocaleLowerCase().indexOf(substr.toLocaleLowerCase());

2.4.4.3 String Built-In Methods

split() divides a string into an array of substrings:

```
split([separator, [,limit]]);
```

The separator determines where each split should occur in the original string. The separator can be a string. Or it can be a regular expression.

Example:

```
let str = 'JavaScript String split()';
let substrings = str.split(' ');
```

2.4.4.4 String Built-In Methods

Example we want to split the string by comma or space

We could take the advantage of using regex in separator

```
const test = "abf,3er ert";
let array = str.split(/[ ,]/);
```

Returning a limited number of substrings

```
let array = str.split(/[ ,]/, 2);
```

2.4.4.5 String Built-In Methods

The includes() method determines whether a string contains another string

```
string.includes(searchString [,position])
```

```
let email = 'admin@example.com’;

console.log(email.includes('@'));
```

Returning a limited number of substrings

```
let array = str.split(/[ ,]/, 2);
```

www.chuwaamerica.com

2.4.4.6 String Built-In Methods

The slice() method returns a substr from the startIndex to the endIndex in the str:

```
let substr = str.slice(startIndex [, endIndex ]);
```

FYI, [startIndex, endIndex), and we have the same built-in methods in Array

www.chuwaamerica.com

2.4.5 Data Types -Boolean

The boolean type has two values: true and false, in lowercase.

To convert a value of another data type into a boolean value, you use the Boolean function.

```
console.log(Boolean('Hi'));// true
```

2.4.6 Data Types -Object

In JavaScript, an object is a collection of properties, where each property is defined as a key-value pair.

The following example defines an empty object using the object literal form:

```
let emptyObject = {};
```

The example defines the person object with two properties:

```
let person = {
            firstName: 'John’,
            lastName: 'Doe’
};
```

2.4.6.1 Data Types – Object attribute rule

A property name of an object can be any string. You can use quotes around the property name if it isn't a valid JavaScript identifier.

For example, if you have a property first-name, you must use the quotes such as "first-name" but firstName is a valid JavaScript identifier so the quotes are optional.

If you refer to a non-existent property, you'll get an undefined value as follows:

```
console.log(contact.age); // undefined
```

2.4.6.2 Data Types – More ways to create object in JavaScript

1. Use new Object to create an object

```
let customer = new Object();
customer.name = 'ABC Inc.';
```

2. Utilize class to create an object

2.5.1 Standard built-in objects - Array

JavaScript provides you with two ways to create an array. The first one is to use the Array constructor as follows

```
let scores = new Array();
```

If you know the number of elements that the array will hold, you can create an array with an initial size as shown in the following example:

```
let scores = Array(10);
```

The more preferred way to create an array is to use the array literal notation:

```
let arrayName = [element1, element2, element3, ...];
```

2.5.1.1 Standard built-in objects – Array 2

Accessing JavaScript array elements
`arrayName[index]`

Getting the array size, using the length property
`arrayName.length`

Adding an element to the end of an array:
`arrayName.push('Red Sea');`

Adding an element to the end of an array(ES6 Syntax):
`arrayName  = […arrayName, 'Red Sea']`

Check if an value is an array
`Array.isArray(seas);`

2.5.1.3 Standard built-in objects – Array 3

Adding an element to the beginning of an array
arrayName.unshift('Red Sea');

Removing an element from the end of an array
arrayName.pop();

Removing an element from the beginning of an array
arrayName.shift();

FYI: could take it as stack or Queue

Finding an index of an element in the array
arrayName.indexOf('Red Sea');

2.5.1.4 Standard built-in objects – Array 4

Deleting elements using JavaScript Array’s splice() method
arrayName.splice(position, num);

Inserting elements using JavaScript Array splice() method
arrayName.splice(position, 0 ,new_element_1,new_element_2,...);

Note that the splice() method actually changes the original array. Also, the splice() method does not remove any elements, therefore, it returns an empty array.

Replacing elements using JavaScript Array splice() method
let languages = ['C', 'C++', 'Java', 'JavaScript’];
languages.splice(1, 1, 'Python');

2.5.1.5 Standard built-in objects – Array 5

The Array.prototype object provides the slice() method that allows you to extract subset elements of an array and add them to the new array

The slice() method accepts two optional parameters as follows:
slice(start, stop); // Both start and stop parameters are optional.
FYI: [start, stop). The slice() returns a new array, it doesn't change the source array

1. Clone an array

numbers.slice();

ES6 Syntax:
const newNumbers = [… numbers];

2.5.1.6 Standard built-in objects – Array 6

Copy a portion of an array

```
const colors = ['red','green','blue','yellow','purple’];
const rgb = colors.slice(0,3);
```

Convert array-like objects into arrays
Take the advantage of call method

www.chuwaamerica.com

2.5.1.7 Standard built-in objects – Array 7

Checking array element meet requirements ?

1. Use for loop to check
2. Use Array built-in every() method

```
let numbers = [1, 3, 5];
for(let i=0; i<numbers.length; i++){
        //code here
}
```

```
numbers.every((ele, index, numbers) => {
  return ele > 0;
});
```

In JavaScript, a callback is a function passed into another function as an argument to be executed later.

Caution: Empty arrays:

the method will always return true for any condition

www.chuwaamerica.com

2.5.1.8 Standard built-in objects – Array 8

Checking at least one elements meet requirements ?

1. Use for loop to check
2. Use Array built-in some() method

```
let numbers = [1, 3, 5];
for(let i=0; i<numbers.length; i++){
        //code here
}

numbers.some((ele, index, numbers) => {
   return ele > 0;
});
```

Caution: Empty arrays:

the method will always return false for any condition

www.chuwaamerica.com

2.5.1.9 Standard built-in objects – Array 9

execute a function on every element of an array

1. Use for loop
2. Use forEach

```
let numbers = [1, 3, 5];
for(let i=0; i<numbers.length; i++){
            //code here
}

numbers.forEach((ele, index, numbers) => {
    // code here
});
```

Question: What is the difference between for loop and forEach ?

2.5.1.10 Standard built-in objects – Array 10

transform its elements and include the results in a new array.

1. Use for loop
2. Use map mthod

```
let numbers = [1, 3, 5];
for(let i=0; i<numbers.length; i++){
        //code here
}

numbers.map((ele, index, numbers) => {
   // code here
});
```

Question: What is the difference between for map and forEach ?

2.5.1.10 Standard built-in objects – Array 11

Reducing an Array Into a Value.

1. Use for loop
2. Use reduce mthod

```
let numbers = [1, 3, 5];
for(let i=0; i<numbers.length; i++){
        //code here
}
```

```
numbers.reduce((accumulator, current, index, numbers) => {
   // code here
   //example: return accumulator + current
}, initValue);
```

2.5.1.12 Standard built-in objects – Array 12

Classical interview question

How to implement a reducer by your own code ?

```
const myReduce = (array, callback, initValue) => {
        let accumulator = initValue === undefined ? 0 : initValue;
        for (let i = 0; i < array.length; i++) {
                accumulator = callback(accumulator, array[i], i, array);
        }
        return accumulator;
};
```

2.5.1.13 Standard built-in objects – Array 13

Use the JavaScript Array filter() method to filter elements in an array

1. Use for loop to check
2. Use Array built-in every() method

```
let numbers = [1, 3, 5];
for(let i=0; i<numbers.length; i++){
        //code here
}

numbers. filter((ele, index, numbers) => {
  return ele > 0;
});
```

www.chuwaamerica.com

2.5.1.13 Standard built-in objects – Array 14

Array join() method to concatenate all elements of an array into a string separated by a separator

1. Use for loop to check
2. Use Array built-in every() method

```
let numbers = [1, 3, 5];
for(let i=0; i<numbers.length; i++){
        //code here
}

const cssClasses = ['btn', 'btn-primary', 'btn-active’];
const btnClass = cssClasses.join(' ');
```

2.5.1.13 Standard built-in objects – Array 14

Array join() method to concatenate all elements of an array into a string separated by a separator

1. Use for loop to check
2. Use Array built-in every() method

```
let numbers = [1, 3, 5];
for(let i=0; i<numbers.length; i++){
            //code here
}

const cssClasses = ['btn', 'btn-primary', 'btn-active’];
const btnClass = cssClasses.join(' ');
```

www.chuwaamerica.com

2.5.1.13 Standard built-in objects – Array 15

Using the JavaScript Array join() method to replace all occurrences of a string

```
const title = 'JavaScript array join example';
const url = title.split(' ') .join('-') .toLowerCase();

console.log(url);
```

2.6.1 JavaScript Function

Declaring functions

you use the function keyword, followed by the function name, a list of parameters, and the function body

```
function functionName(parameters) {
    // function body // ...
}
```

ES6 Syntax

```
const functionName = (parameters) => {
            // function body // ...
}
```

2.6.2 Function hoisting 2

In JavaScript, it is possible to use a function before it is declared. See the following example:

```
showMe();  // call no matte the declaration position

showMe(){
    console.log('an hoisting example');
}
```

How about we use ES6 syntax to implement a function ?

2.6.3 Function hoisting

If we call the function as the following code

```
functionName();

const functionName = (parameters) => {
        // function body // ...
}
```

It doesn't work !

How about the following code ?

```
typeof functionName;

var functionName = (parameters) => {
        // function body // ...
}
```

Same as the variable hoisting

www.chuwaamerica.com

2.6.4 The arguments object

Inside the body of a function, you can access an object called arguments that represents the named arguments of the function.

The arguments object behaves like an array though it is not an instance of the Array type.

For example, you can use the square bracket [] to access the arguments: arguments[0] returns the first argument, arguments[1] returns the second one, and so on.

```
function add(para1, para2 ….) {
   // return the sum of the arguments
}
```

2.7.1 DOM manipulation

The Document Object Model (DOM) is an application programming interface (API) for manipulating HTML and XML documents.

The DOM represents a document as a tree of nodes. It provides API that allows you to add, remove, and modify parts of the document effectively.

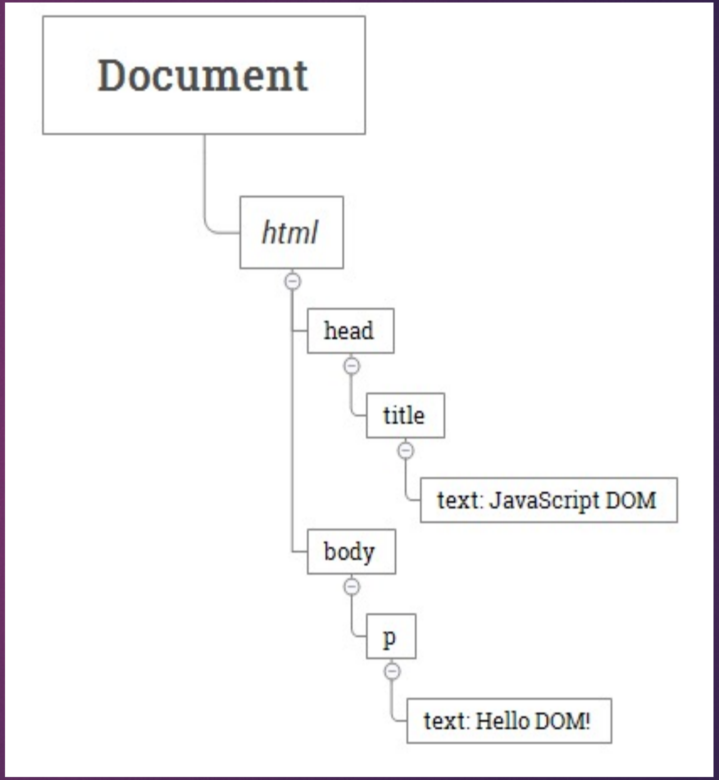
In this DOM tree, the document is the root node. The root node has one child which is the <html> element. The <html> element is called the document element.

www.chuwaamerica.com

2.7.2 DOM manipulation

HTML Code

```
<html>
  <head>
    <title>JavaScript DOM</title>
  </head>
  <body>
    <p>Hello DOM!</p>
  </body>
</html>
```



2.7.3.1 Selecting elements - getElementById

The following shows the syntax of the getElementById() method:

```
let element = document.getElementById(id);
```

Once you selected an element, you can add styles to the element, manipulate its attributes

www.chuwaamerica.com

2.7.3.2 Selecting elements - getElementsByName()

The following shows the syntax of the getElementByName() method:

```
<input type=”text" name="language" value="JavaScript">
<input type=”text" name="language" value=”Java">

let elements = document.getElementByName(”language" );
```

The getElementsByName() accepts a name which is the value of the name attribute of elements and returns a live NodeList of elements.

Note: Although NodeList is not an Array, it is possible to iterate over it with forEach(). It can also be converted to a real Array using Array.from().

www.chuwaamerica.com

2.7.3.3 Selecting elements – others similar select methods

getElementsByTagName()
getElementsByClassName()

FYI: If you match elements by multiple classes, you need to use whitespace to separate them like this:

Example:

```
let btn = document.getElementsByClassName('btn bt-primary’);
```

The return type: HTMLCollection

2.7.3.4 Selecting elements – querySelector and querySelectorAll

The querySelector() is a method of the Element interface. The querySelector() allows you to find the first element that matches one or more CSS selectors.

Example:

```
let firstHeading = document.querySelector('h1’);
let note = document.querySelector('.menu-item’);
let logo = document.querySelector('#logo’);
```

querySelectorAll() method to find all elements that match a CSS selector or a group of CSS selector.

querySelectorAll() returns a static (not live) NodeList representing a list of the document's elements that match the specified group of selectors.

www.chuwaamerica.com

2.7.5 Manipulating elements - createElement

To create an HTML element, you use the document.createElement() method

```
let div = document.createElement('div');

div.id = 'content’;
div.className = 'note’;
div.textContent= Thi s is a Test’;

document.body.appendChild(div)
```

2.7.6 Manipulating elements - appendChild()

The appendChild() method allows you to add a node to the end of the list of child nodes of a specified parent node;

```
parentNode.appendChild(childNode);
```

```
function createMenuItem(name) {
  let li = document.createElement('li’);
  li.textContent = name;
  return li;
}
```

```
const menu = document.querySelector('#menu’);
//dynamic add element to list
menu.appendChild(createMenuItem('Home'));
menu.appendChild(createMenuItem('Services'));
menu.appendChild(createMenuItem('About Us'));
```

2.7.7 Manipulating elements - textContent

To get the text content of a node and its descendants, you use the textContent property

```
<div id="note">
        JavaScript textContent Demo!
        <span style="display:none">Hidden Text!</span>
         <!-- my comment -->
</div>

let note = document.getElementById('note’);
console.log(note.textContent);
```

2.7.7.1 Manipulating elements - textContent

Setting textContent for a node

When you set textContent on a node, all the node's children will be removed and replaced by a single text node with the newText value. For example:

```
let note = document.getElementById('note');
note.textContent = 'This is a note';
```

2.7.8 Manipulating elements - innerHTML

To get the HTML markup contained within an element, you use the following syntax

```
<ul id="menu">
    <li>Home</li>
    <li>Services</li>
</ul>
```

The following example uses the innerHTML property to get the content of the <ul> element:

```
let menu = document.getElementById('menu’);
console.log(menu.innerHTML);
```

2.7.8.1 Manipulating elements – innerHTML Security Risk

```
<ul id=”app">
</ul>

let app = document.getElementById(‘app’);
app.innerHTML = `<img src='1' onerror='alert("Error loading image")’>`;
```

Because of this, if you want to insert plain text into the document, you use the textContent property instead of the innerHTML. The textContent will not be parsed as the HTML, but as the raw text.

www.chuwaamerica.com

2.7.9 Manipulating elements – insertBefore

To insert a node before another node as a child node of a parent node, you use the parentNode.insertBefore()

```
parentNode.insertBefore(newNode, existingNode);
```

- The newNode is the new node to be inserted.
- The existingNode is the node before which the new node is inserted. If the existingNode is null, the insertBefore() inserts the newNode at the end of the parentNode's child nodes.

2.7.9.1 Manipulating elements – insertBefore example

```
<ul id="menu">
   <li>Services</li>
   <li>About</li>
  <li>Contact</li>
</ul>

let menu = document.getElementById('menu’);
// create a new li node
let li = document.createElement('li’);
li.textContent = 'Home’;

// insert a new node before the first list item
menu.insertBefore(li, menu.firstElementChild);
```

www.chuwaamerica.com

2.7.10 Manipulating elements – insertAfter

JavaScript DOM provides the insertBefore() method that allows you to insert a new after an existing node as a child node. However, it hasn’t supported the insertAfter() method yet.

We could implement a helper function to achieve the goal

```
function insertAfter(newNode, existingNode){
    existingNode.parentNode.insertBefore(newNode, existingNode.nextSibling);
 }
```

2.7.10.1 Manipulating elements – insertAfter example

```
<ul id="menu">
   <li>Services</li>
   <li>About</li>
  <li>Contact</li>
</ul>

let menu = document.getElementById('menu’);
// create a new li node
let li = document.createElement('li’);
li.textContent = 'Services’;
// insert a new node after the first list item
menu.insertBefore(li, menu.firstElementChild.nextSibling);
```

2.7.11 Manipulating elements – append

The parentNode.append() method inserts a set of Node objects or DOMString objects after the last child of a parent node:

parentNode.append(...nodes);

The append() method has no return value.
It means that the append() method implicitly returns undefined.

2.7.11.1 Manipulating elements – append example

```
<ul id="app">
    <li>JavaScript</li>
</ul>

let app = document.querySelector('#app');
let langs = ['TypeScript','HTML','CSS'];

let nodes = langs.map(lang => {
    let li = document.createElement('li');
    li.textContent = lang;
    return li;
});

app.append(...nodes);
```

2.7.11.2 Manipulating elements – append and appendChild

Differences	append()	appendChild()
Return value	undefined	The appended Node object
Input	Multiple Node Objects	A single Node object
Parameter Types	Accept Node and DOMString	Only Node

2.7.12 Manipulating elements – prepend

The prepend() method inserts a set of Node objects or DOMString objects after the first child of a parent node:

```
parentNode.prepend(...nodes);
parentNode.prepend(...DOMStrings);
```

The prepend() method returns undefined.

2.7.12.1 Manipulating elements – prepend example

```
<ul id="app">
    <li>HTML</li>
</ul>

let app = document.querySelector('#app');
let langs = ['CSS','JavaScript','TypeScript'];
let nodes = langs.map(lang => {
    let li = document.createElement('li');
    li.textContent = lang;
    return li;
});
app.prepend(...nodes);
```

www.chuwaamerica.com

2.7.13 Manipulating elements – replaceChild

To replace an HTML element, you use the node.replaceChild() method

```
parentNode.replaceChild(newChild, oldChild);
```

In this method, the newChild is the new node to replace the oldChild node which is the old child node to be replaced.

2.7.13.1 Manipulating elements – replaceChild example

```
<ul id="menu">
   <li>Homepage</li>
   <li>Services</li>
   <li>About</li>
   <li>Contact</li>
</ul>

let menu = document.getElementById('menu');
// create a new node
let li = document.createElement('li');
li.textContent = 'Home';
// replace the first list item

menu.replaceChild(li, menu.firstElementChild);
```

2.7.14 Manipulating elements – removeChild

To remove a child element of a node, you use the removeChild() method

let childNode = parentNode.removeChild(childNode);

The childNode is the child node of the parentNode that you want to remove.
If the childNode is not the child node of the parentNode, the method throws an exception.

The removeChild() returns the removed child node from the DOM tree but keeps it in the memory, which can be used later.

You can directly use: parentNode.removeChild(childNode);

www.chuwaamerica.com

2.7.14.1 Manipulating elements – removeChild example

```
<ul id="menu">
   <li>Home</li>
   <li>Products</li>
   <li>About Us</li>
</ul>

let menu = document.getElementById('menu');
menu.removeChild(menu.lastElementChild);
```

How to remove all child nodes of an element ?

2.7.14.2 Manipulating elements – removeChild example

```
let menu = document.getElementById('menu');
while (menu.firstChild) {
    menu.removeChild(menu.firstChild);
}
```

2.8.1 Working with Events – Intro to Event

What is an event ?

An event is an action that occurs in the web browser, which the web browser feedbacks to you so that you can respond to it.

Each event may have an event handler which is a block of code that will execute when the event occurs.

An event handler is also known as an event listener. It listens to the event and executes when the event occurs.

2.8.2 Working with Events– Events Examples

```
<button id="btn">Click Me!</button>

let btn = document.querySelector('#btn');

btn.addEventListener('click',function() {
    alert('It was clicked!');
});
```

Or

```
<button id=“btn” onclick=‘alert('It was clicked!');’>Click Me!</button>
```

It’s a good practice to always use the addEventListener() to register a mouse event handler. This rule is for vanilla JavaScript only.

www.chuwaamerica.com

2.8.2.1 Working with Events– Disadvantages of event handler attributes

Assigning event handlers using HTML event handler attributes are considered as bad practices and should be avoided as much as possible because of the following reasons:

First, the event handler code is mixed with the HTML code, which will make the code more difficult to maintain and extend.

Second, it is a timing issue. If the element is loaded fully before the JavaScript code, users can start interacting with the element on the webpage which will cause an error.

2.8.2.2 Working with Events – event handlers Level One

- DOM Level 0 event handlers

```
let btn = document.querySelector('#btn');

btn.onclick = function() {
    alert(this.id);
};
```

To remove the event handler, you set the value of the event handler property to null:

```
btn.onclick = null;
```

The DOM Level 0 event handlers are still being used widely because of its simplicity and cross-browser support.

www.chuwaamerica.com

2.8.2.2 Working with Events – event handlers Level Two

- DOM Level 2 event handlers

DOM Level 2 Event Handlers provide two main methods for dealing with the registering/deregistering event listeners:

- addEventListener() – register an event handler
- removeEventListener() – remove an event handler

FYI:
You need to pass the same arguments as were passed to the addEventListener(), using an anonymous event listener will not work.

2.8.3 Working with Events – Events Flow

```
<!DOCTYPE html>
<html>
<head>
   <title>JS Event Demo</title>
</head>
<body>
   <div id="container">
      <button id='btn'>Click Me!</button>
   </div>
</body>
```

When you click the button, you're clicking not only the button but also the button's container, the div, and the whole webpage.

www.chuwaamerica.com

2.8.3.1 Working with Events– Events Bubbling

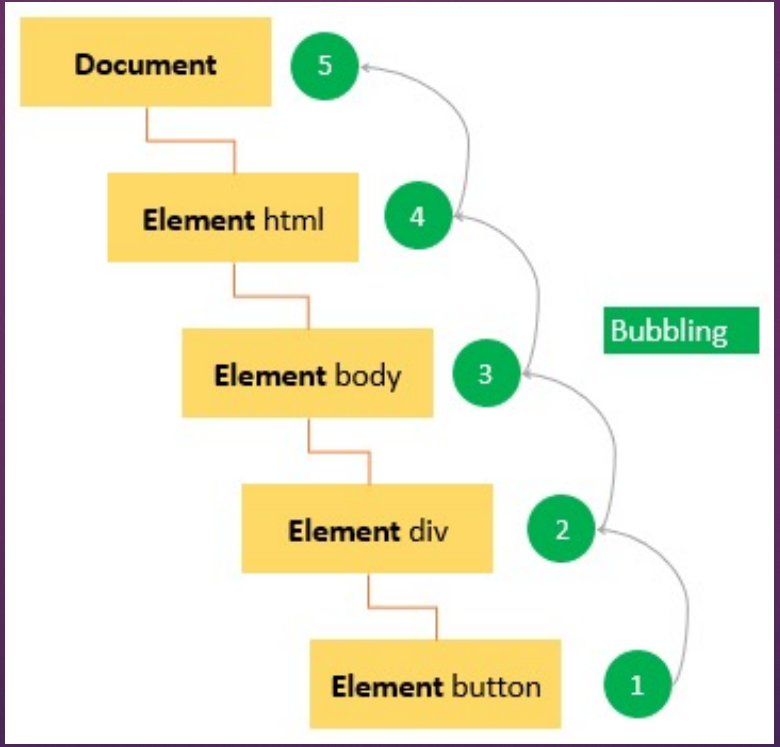
In the event bubbling model, an event starts at the most specific element and then flows upward toward the least specific element

When you click the button, the click event occurs in the following order:

1. button
2. div with the id container
3. body
4. html
5. document

2.8.3.2 Working with Events– Events Bubbling 2

The following picture illustrates the event bubbling effect when users click the button



2.8.3.3 Working with Events – Event capturing

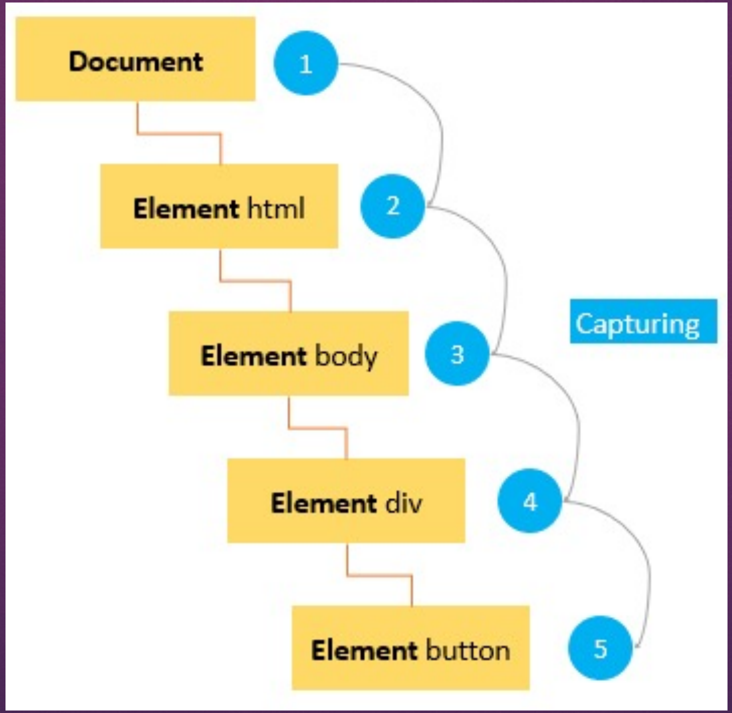
In the event capturing model, an event starts at the least specific element and flows downward toward the most specific element.

When you click the button, the click event occurs in the following order:

1. document
2. html
3. body
4. div with the id container
5. button

2.8.3.4 Working with Events– Event capturing 2

The following picture illustrates the event capturing effect:



2.8.4 Working with Events– Event object

When the event occurs, the web browser passed an Event object to the event handler:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click', function(event) {
    console.log(event.type);
});
```

2.8.4.1 Working with Events – Event object properties and methods

Property / Method	**Description**
bubbles	true if the event bubbles
cancelable	true if the default behavior of the event can be canceled
currentTarget	the current element on which the event is firing
defaultPrevented	return true if the preventDefault() has been called.
detail	more informatio nabout the event
eventPhase	1 for capturing phase, 2 for target, 3 for bubbling
preventDefault()	cancel the default behavior for the event. This method is only effective if the cancelable property is true
stopPropagation()	cancel any further event capturing or bubbling. This method only can be used if the bubbles property is true.
target	the target element of the event
type	the type of event that was fired

2.8.4.2 Working with Events – preventDefault()

To prevent the default behavior of an event, you use the preventDefault() method. For example, when you click a link, the browser navigates you to the URL specified in the href attribute:

```
<a href="https://www.chuwaamerica.com/">Front End Training</a>

let link = document.querySelector('a');

link.addEventListener('click',function(event) {
    console.log('clicked');
    event.preventDefault();
});
```

2.8.4.3 Working with Events – stopPropagation()

The stopPropagation() method immediately stops the flow of an event through the DOM tree. However, it does not stop the browers default behavior.

```
let btn = document.querySelector('#btn');

btn.addEventListener('click', function(event) {
   console.log('The button was clicked!');
   event.stopPropagation(); // add later
});

document.body.addEventListener('click',function(event) {
   console.log('The body was clicked!');
});
```

www.chuwaamerica.com

2.8.5 Working with Events – Event Delegation

```
<ul id="menu">
    <li><a id="home">home</a></li>
    <li><a id="dashboard">Dashboard</a></li>
    <li><a id="report">report</a></li>
</ul>
```

To handle the click event of each menu item, you may add the corresponding click event handlers:

```
let home = document.querySelector('#home');
home.addEventListener('home',(event) => {
    console.log('Home menu item was clicked');
});
```

let dashboard and let report…….(same logic as the code above)

2.8.5.1 Working with Events – Event Delegation 2

In JavaScript, if you have a large number of event handlers(more than 200) on a page, these event handlers will directly impact the performance because of the following reasons:

- First, each event handler is a function which is also an object that takes up memory. The more objects in the memory, the slower the performance.
- Second, it takes time to assign all the event handlers, which causes a delay in the interactivity of the page.

Event Delegation(take the advantage of event bubbling) can help !

2.8.5.1 Working with Events – Event Delegation Example

Instead of having multiple event handlers, you can assign a single event handler to handle all the click events:

```
let menu = document.querySelector('#menu');
menu.addEventListener('click', (event) => {
    let target = event.target;
    switch(target.id) {
        case 'home':
            console.log('Home menu item was clicked');
            break;
        case 'dashboard':
            console.log('Dashboard menu item was clicked');
            break;
        case 'report':
            console.log('Report menu item was clicked');
            break;
});
```

2.9 JavaScript Advanced Topic - Anonymous Functions

An anonymous function is a function without a name. An anonymous function is often not accessible after its initial creation.

```
let show = function () {
    console.log('Anonymous function');
};

show();
```

In this example, the anonymous function has no name between the function keyword and parentheses ().

We often use anonymous functions as arguments of other functions. For example: Callback function

2.9.1 JavaScript Advanced Topic - immediately invoked function

A JavaScript immediately invoked function expression(IIFE) is a function defined as an expression and executed immediately after creation

```
(function(){
   //...
})();
```

ES6 syntax:

```
(() => {
   //...
})();
```

How about giving a name to IIFE ? it cannot be invoked again after execution

www.chuwaamerica.com

2.9.2 JavaScript Advanced Topic - Closure

A closure gives you access to an outer function's scope from an inner function.

```
function greeting() {
   let message = 'Hi';

   function sayHi() {
      console.log(message);
   }
   return sayHi;
}
let hi = greeting();
hi(); // still can access the message variable
```

2.9.2.1 JavaScript Advanced Topic - Closure Examples

```
function greeting(message) {
  return function(name){
      return message + ' ' + name;
  }
}
let sayHi = greeting('Hi');
let sayHello = greeting('Hello');

console.log(sayHi('John')); // Hi John
console.log(sayHello('John')); // Hello John
```

How about using ES6 syntax for the example above?

www.chuwaamerica.com

2.9.2.2 JavaScript Advanced Topic - Closure Examples

The greeting() function behaves like a function factory. It
creates sayHi() and sayHello() functions with the respective messages Hi and Hello.

The sayHi() and sayHello() are closures. They share the same function body but store different scopes.

How about using ES6 syntax for the example above?

www.chuwaamerica.com

2.9.2.3 JavaScript Advanced Topic - Closure Examples 2

```
for (var index = 1; index <= 3; index++) {
    setTimeout(function () {
        console.log('after ' + index + ' second(s):' + index);
    }, index * 1000);
}
```

What is the output ?

www.chuwaamerica.com

2.9.2.4 JavaScript Advanced Topic - Closure Examples 2

```
after 4 second(s):4
after 4 second(s):4
after 4 second(s):4
```

Why It happened ?

1. Use var to declare variable(non block scope) will be considered as a global variable
2. It is an asynchronous task(setTimeout), the asynchronous tasks will be put into the task queue.
3. When the synchronous tasks are completed, the asynchronous tasks will be executed(setTimeout)

2.9.2.5 JavaScript Advanced Topic - Closure Examples 3

How to fix it ?

- Using the IIFE solution

```
for (var index = 1; index <= 3; index++) {
    (function (index) {
        setTimeout(function () {
            console.log('after ' + index + ' second(s):' + index);
        }, index * 1000);
    })(index);
}
```

an IIFE creates a new scope by declaring a function and immediately execute it.

www.chuwaamerica.com

2.9.2.6 JavaScript Advanced Topic - Closure Examples 4

- Using let keyword in ES6

```
for (let index = 1; index <= 3; index++) {
    setTimeout(function () {
        console.log('after ' + index + ' second(s):' + index);
    }, index * 1000);
}
```

it will create a new lexical scope in each iteration. In other words, you will have a new index variable in each iteration.

www.chuwaamerica.com

2.9.2.7 JavaScript Advanced Topic - Closure Examples 5

How to create a counter for counting the function’s execution time ?

2.9.2.8 JavaScript Advanced Topic - Closure Examples 6

Write a sum method which will work properly when invoked using either syntax below.

```
console.log(sum(2,3)); // Outputs 5
console.log(sum(2)(3)); // Outputs 5
```

2.9.2.8 JavaScript Advanced Topic - Closure Examples 7

Consider the code snippet below. What will the console output be and why?

```
(function (x) {
  return (function (y) {
    console.log(x);
  })(2);
})(1);
```

2.9.3.1 JavaScript Advanced Topic - passing by value

Passing by value of primitives values, string, Boolean and number

```
function square(x) {
    x = x * x;
    return x;
}
var y = 10;
var result = square(y);
console.log(y); // 10 -- no change
console.log(result); // 100
```

2.9.3.2 JavaScript Advanced Topic - Passing by value of object

Passing by value of object

```
function turnOn(machine) {
    machine.isOn = true;
}

var computer = {
    isOn: false
};

turnOn(computer);
console.log(computer.isOn); // true;
```

2.9.4 JavaScript Advanced Topic - call() method

In JavaScript, a function is an instance of the Function type. For example:

```
function show(arg1, arg2){
    //...
}

console.log(show instanceof Function); // true
```

The Function type has a method named call() with the following syntax:

```
functionName.call(thisArg, arg1, arg2, ...);
```

2.9.4.1 JavaScript Advanced Topic - call() method 2

The first argument of the call() method thisArg is the this value. It allows you to set the this value to any given object.

When you invoke a function, the JavaScript engine invokes the call() method of that function object.

```
function show() {
   console.log('Show function');
}

show();

=>  show.call();
```

2.9.4.2 JavaScript Advanced Topic - call() method 3

By default, the this value inside the function is set to the global object i.e., window on web browsers and global on node.js:

```
function show() {
    console.log(this);
}

show();
```

Note that in the strict mode, the this inside the function is set to undefined instead of the global object.

2.9.5 JavaScript Advanced Topic - apply() method

The Function.prototype.apply() method allows you to call a function with a given this value and arguments provided as an array. Here is the syntax of the apply() method:

```
fn.apply(thisArg, [args]);
```

The apply() method accepts two arguments:
The thisArg is the value of this provided for the call to the function fn.
The args argument is an array that specifies the arguments of the function fn. Since the ES5, the args argument can be an array-like object or array object.

THe apply() method is similar to the call() method except that it takes the arguments of the function as an array instead of the individual arguments.

2.9.5.1 JavaScript Advanced Topic - apply() method 2

Suppose that you have a person object:

```
const person = {
    firstName: 'John',
    lastName: 'Doe’
}

function greet(greeting, message) {
    return `${greeting} ${this.firstName}. ${message}`;
}

greet(‘Hi’, ‘How are you’)  // vs
greet.apply(person, ['Hi', 'How are you'])
```

www.chuwaamerica.com

2.9.5.2 JavaScript Advanced Topic - apply() method 3

The apply() method allows you to append elements of an array to another:

```
let arr = [1, 2, 3];
let numbers = [4, 5, 6];

arr.push.apply(arr, numbers);

console.log(arr);
```

2.9.6 JavaScript Advanced Topic - bind() method

The bind() method returns a new function, when invoked, has its this sets to a specific value.
The following illustrates the syntax of the bind() method:

```
fn.bind(thisArg[, arg1[, arg2[, ...]]])
```

Unlike the call() and apply() methods, the bind() method doesn't immediately execute the function. It just returns a new version of the function whose this sets to thisArg argument.

2.9.6.1 JavaScript Advanced Topic - bind() method 2

```
let person = {
    name: 'John Doe',
    getName: function() {
        console.log(this.name);
    }
};

setTimeout(person.getName, 1000);
```

Output : undefined,

Why?

2.9.6.2 JavaScript Advanced Topic - bind() method 3

As you can see clearly from the output, the person.getName() returns undefined instead of 'John Doe'.

This is because setTimeout() received the function person.getName separately from the person object.

```
setTimeout(person.getName, 1000);

=>

let f = person.getName;
setTimeout(f, 1000); // lost person context
```

2.9.6.3 JavaScript Advanced Topic - bind() method 4

1. One fix solution:

```
setTimeout(function () {
    person.getName();
}, 1000);
```

ES6 syntax ?

2. Use bind method

```
let f = person.getName.bind(person);
setTimeout(f, 1000);
```

2.9.7 JavaScript Advanced Topic – Call, Apply and Bind

All three of these JavaScript methods allow you to change the value of this for a given function.

Call invokes the function and allows you to pass in arguments one by one.

Apply invokes the function and allows you to pass in arguments as an array.

Bind returns a new function, then you can execute this function by passing the arguments.

www.chuwaamerica.com

2.10 JavaScript Advanced Topic – JavaScript Fetch API

The fetch() requires only one parameter which is the URL of the resource that you want to fetch:

```
let response = fetch(url);
```

The fetch() method returns a Promise so you can use the then() and catch() methods to handle it:

```
fetch(url)
    .then(response => {
        // handle the response
    })
    .catch(error => {
        // handle the error
    });
```

2.10.1 JavaScript Advanced Topic – JavaScript Fetch API 2

What is promise ?

In JavaScript, a promise is an object that returns a value which you hope to receive in the future, but not now.

Because the value will be returned by the promise in the future, the promise is very well-suited for handling asynchronous operations.

A promise has three states:

- Pending: you don't know if you will complete learning JavaScript by the next month.
- Fulfilled: you complete learning JavaScript by the next month.
- Rejected: you don't learn JavaScript at all.

2.10.2 JavaScript Advanced Topic – JavaScript Fetch API 3

To create a promise in JavaScript, you use the Promise constructor:

```
let completed = true;

let learnJS = new Promise(function (resolve, reject) {
    if (completed) {
        resolve("I have completed learning JS.");
    } else {
        reject("I haven't completed learning JS yet.");
    }
});
```

2.10.4 JavaScript Advanced Topic – JavaScript Fetch API 5

To see the pending state of the promise, we wrap the code of the executor in the setTimeout() function:

```
let completed = true;
let learnJS = new Promise(function (resolve, reject) {
    setTimeout(() => {
        if (completed) {
            resolve("I have completed learning JS.");
        } else {
            reject("I haven't completed learning JS yet.");
        }
    }, 3 * 1000);
});
```

2.10.5 JavaScript Advanced Topic – JavaScript Fetch API 5

Async and wait

An async function is a function declared with the async keyword,
and the await keyword is permitted within
them. The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

Async and wait should be used in together.

2.10.6 JavaScript Advanced Topic – JavaScript Fetch API 6

Interview code challenge:

Given an API, consuming the data from backend and create the list dynamically by vanilla JavaScript?

2.11 JavaScript Advanced Topic – Event Loop

JavaScript single-threaded model

JavaScript is a single-threaded programming language. In other words, JavaScript can do only one thing at a single point in time.

If a function takes a long time to execute, you cannot interact with the web browser during the function's execution because the page hangs.

A function that takes a long time to complete is called a blocking function. Technically, a blocking function blocks all the interactions of the webpage, such as mouse click.

www.chuwaamerica.com

2.11.1 JavaScript Advanced Topic – Event Loop 2

```
function task(message) {
    // emulate time consuming task
    let n = 10000000000;
    while (n > 0){
        n--;
    }
    console.log(message);
}

console.log('Start script...');
task('Download a file.');
console.log('Done!');
```

2.11.2 JavaScript Advanced Topic – Event Loop 3

In this example, we have a big while loop inside the task() function that emulates a time-consuming task. The task() function is a blocking function.

The script hangs for a few seconds and issues the following output:

```
Start script...
Download a file.
Done!
```

2.11.3 JavaScript Advanced Topic – Event Loop 4

Callbacks to the rescue

To prevent a blocking function from blocking other activities, you typically put it in a callback function for execution later. For example:

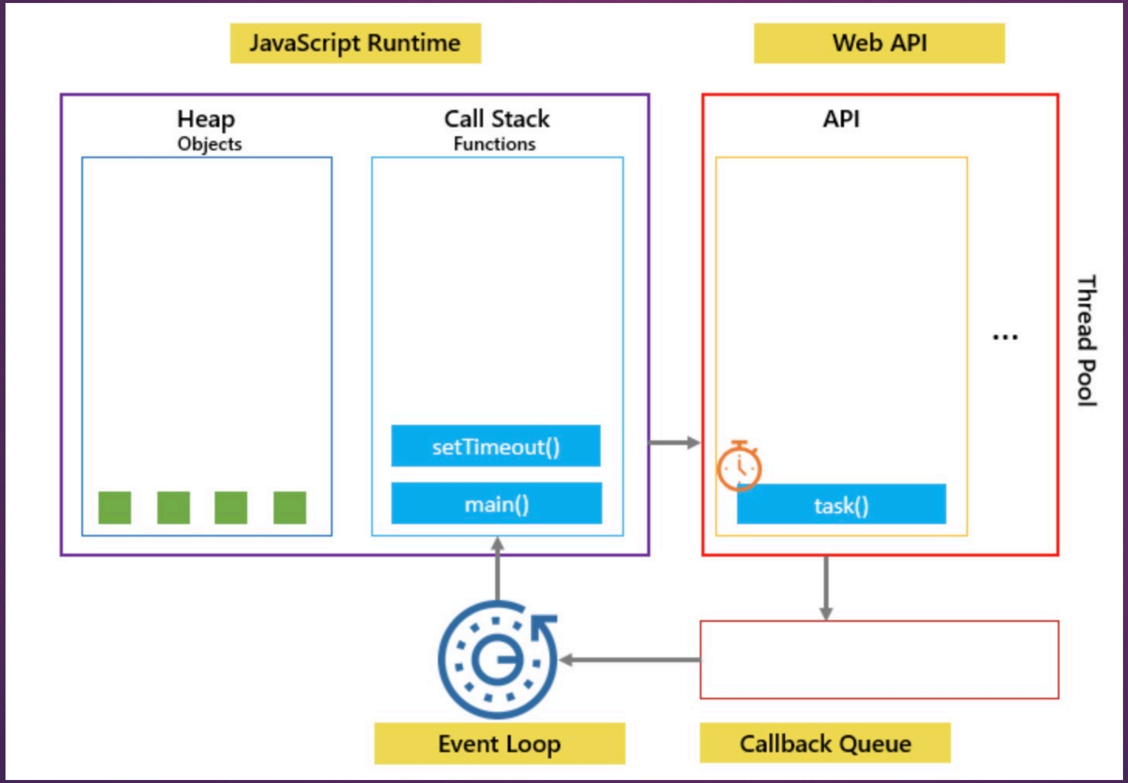
```
console.log('Start script...');

setTimeout(() => {
    task('Download a file.');
}, 1000);

console.log('Done!');
```

2.11.4 JavaScript Advanced Topic – Event Loop 5

When you call the setTimeout() function, make a fetch request or click a button, the web browser can do these activities concurrently and asynchronously.

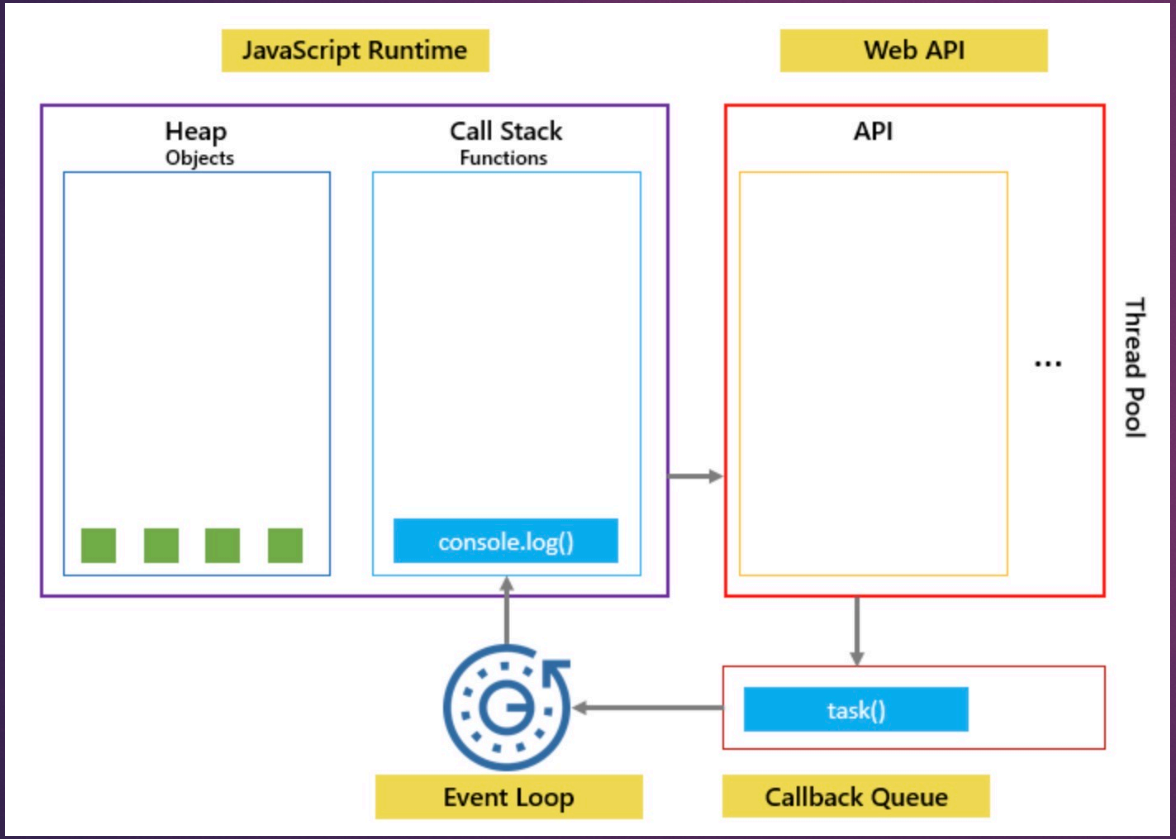
when you call the setTimeout() function, the JavaScript engine places it on the call stack, and the Web API creates a timer that expires in 1 second.

2.11.5 JavaScript Advanced Topic – Event Loop 6



www.chuwaamerica.com

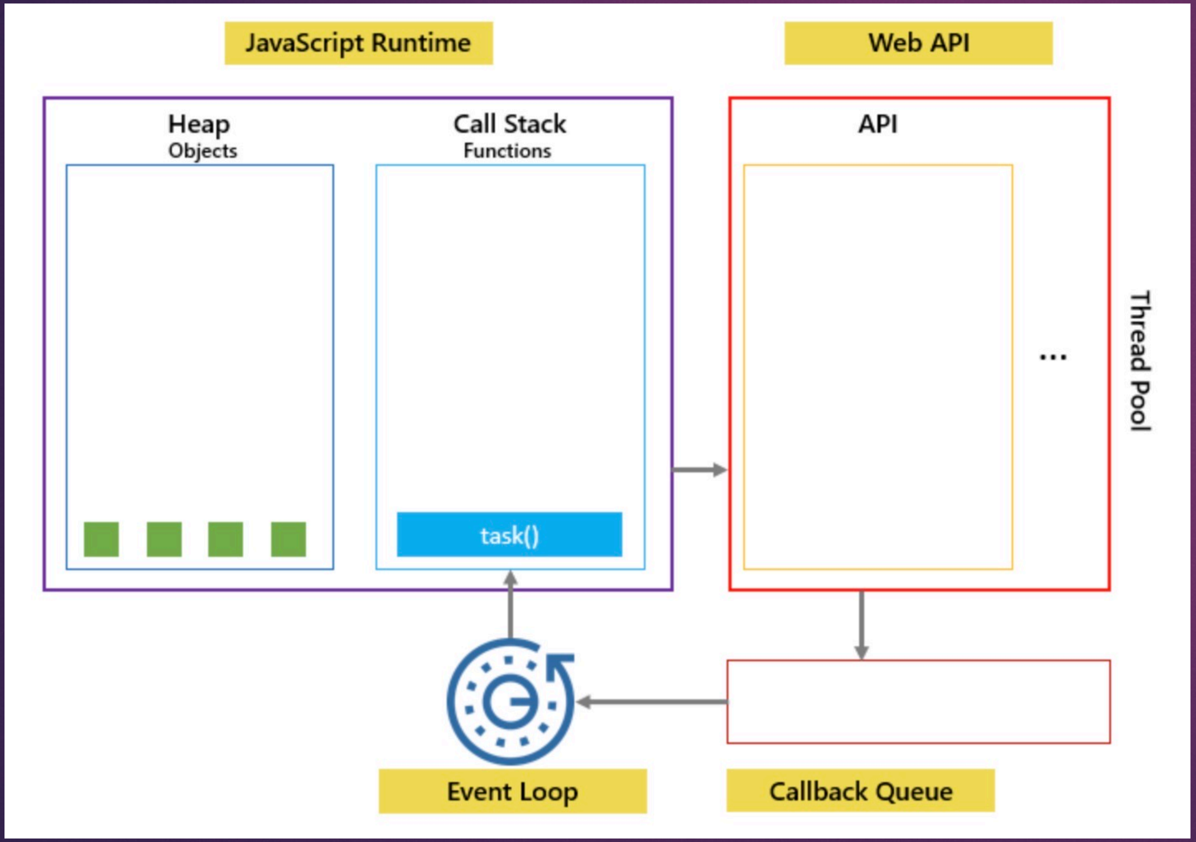
2.11.6 JavaScript Advanced Topic – Event Loop 7



Then JavaScript engine place the task() function is into a queue called a callback queue or a task queue:

www.chuwaamerica.com

2.11.7 JavaScript Advanced Topic – Event Loop 8



The event loop is a constantly running process that monitors both the callback queue and the call stack.

If the call stack is not empty, the event loop waits until it is empty and places the next function from the callback queue to the call stack. If the callback queue is empty, nothing will happen:

2.11.8 JavaScript Advanced Topic – Event Loop 9

```
console.log('Hi!');

setTimeout(() => {
    console.log('Execute immediately.');
}, 0);

console.log('Bye!’);
```

The JavaScript engine places the following function call on the callback queue and executes it when the call stack is empty. In other words, the JavaScript engine executes it after the console.log('Bye!').

2.12 JavaScript Advanced Topic – This Keyword

In JavaScript, you can use the this keyword in the global and function contexts. Moreover, the behavior of the this keyword changes between strict and non-strict modes.

What is the this keyword
The this references the object of which the function is a property. In other words, the this references the object that is currently calling the function.

2.12.1 JavaScript Advanced Topic – This Keyword 2

```
const counter = {
   count: 0,
   next: function () {
      return ++this.count;
   }
};

counter.next();
```

Suppose that you have an object called counter. This object counter has a method called next(). When you call the next() method, you can access the this object.

The next() is a function that is the property of the counter object. Therefore, inside the next() function, the this references the counter object

2.12.2 JavaScript Advanced Topic – This Keyword 3

Global context

In the global context, the this references the global object, which is the window object on the web browser or global object on Node.js.

This behavior is consistent whether the strict mode is applied or not, like this

```
console.log(this === window); // true

this.color= 'Red';
console.log(window.color); // 'Red'
```

2.12.3 JavaScript Advanced Topic – This Keyword 4

In JavaScript, you can invoke a function in the following ways:

- Function invocation => simple invocation
- Method invocation => invoke a method in an object
- Constructor invocation => use new key word
- Indirect invocation => use call and apply

www.chuwaamerica.com

2.12.4 JavaScript Advanced Topic – This Keyword 5

non-strict mode:

```
function show() {
   console.log(this === window); //
true
}

show();  // => window.show();
```

"use strict";

```
function show() {
    console.log(this === undefined);
}

show();
```

www.chuwaamerica.com

2.12.5 JavaScript Advanced Topic – This Keyword 6

```
let car = {
    brand: 'Honda',
    getBrand: function () {
        return this.brand;
    }
}

console.log(car.getBrand()); // Honda

let brand = car.getBrand;
console.log(brand());
```

To fix this issue, you use the bind() method

```
let brand = car.getBrand.bind(car);
console.log(brand()); // Honda
```

2.12.6 JavaScript Advanced Topic – This Keyword 7

```
let car = {
    brand: 'Honda',
    getBrand: function () {
        return this.brand;
    }
}

let bike = {
    brand: 'Harley Davidson'
}

let brand = car.getBrand.bind(bike);
console.log(brand());
```

www.chuwaamerica.com

2.12.7 JavaScript Advanced Topic – This Keyword 8

This in Arrow functions

ES6 introduced a new concept named arrow function. In arrow functions, JavaScript sets the this lexically.

It means the arrow function does not create its own execution context but inherits the this from the outer function where the arrow function is defined. See the following example:

```
let getThis = () => this;
console.log(getThis() === window); // true
```

2.12.7 JavaScript Advanced Topic – This Keyword 8

```
function Car() {
    this.speed = 120;
}

Car.prototype.getSpeed = () => {
    return this.speed;
}

var car = new Car();
car.getSpeed(); // TypeError
```

Question : Could I use bind to change the context ?
NO, The call, apply and bind methods are NOT suitable for Arrow functions

www.chuwaamerica.com

2.12.8 JavaScript Advanced Topic – This Keyword 9

```
var myObject = {
 foo: 'bar‘,
 func: function () {
  var self = this;
  console.log('outer func: this.foo = ' + this.foo);
  console.log('outer func: self.foo = ' + self.foo);
  (function () {
   console.log('inner func: this.foo = ' + this.foo);
   console.log('inner func: this = ' + this);
   console.log('inner func: self.foo = ' + self.foo);
  })();
 },
};
myObject.func();
```

2.12 JavaScript Advanced Topic – Summary

Please practice following bullet points:

- All the string, array, and object built-in method eg: forEach, filter, slice, indexOf, map, reduce…
- Closure
- Event Loop
- Async programming
- DOM manipulation(Event bubbling and Event capture)
- This keyword