# Advanced Javascript

# This

**This** keyword refers to the object it belongs to. It has different values depending on where it is used:

- In a method, this refers to the owner object. Ex.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};
```

- Alone, this refers to the global object. Ex.

```
let x = this; // [object Window]
```

- In a regular function, this refers to the global object. Ex.

```
function myFunction() {
  return this;  // [object Window]
}
```

- In a function, in strict mode, this is undefined. Ex.

```
"use strict";
function myFunction() {
```

# This(contd.)

```
  return this; // undefined
}
```

- In an event, this refers to the element that received the event. Ex.

```
<button onclick="this.style.display='none'"> // this refers to the button itself
  Click to Remove Me!
</button>
```

- Methods like call(), apply() and bind() can refer this to any object. Ex.

```
const person1 = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
const person2 = {
  firstName: "John",
  lastName: "Doe",
}
let x = person1.fullName.call(person2); // this in fullName function refers to person2 here
let y = person1.fullName.apply(person2); // this in fullName function refers to person2 here
```

# Hoisting

JS's default behavior of moving **only** declarations to the top of the current scope (to the top of the current script or the current function). In other words; a variable/function can be used before it has been declared. Ex.

```
x = 5; // Assign 5 to x
console.log(x); // 5
var x; // Declare x
-----------------------------------------------------------
console.log(functionBelow("Hello")); // Hello
function functionBelow(greet) {
  return `${greet} world`;
}
```

Variables and constants declared with let or const & arrow functions are not hoisted! Ex.

```
carName = "Volvo"; // ReferenceError
let carName;
```

JavaScript only hoists declarations, not initializations. Ex.

```
var x = 5; // Initialize x
console.log("x is " + x + " and y is " + y); // x is 5 and y is undefined
var y = 7; // Initialize y
```

**Note**: To avoid bugs, always declare all variables at the beginning of every scope.

# Call

call() can be used to invoke (call) a method with an owner object as an argument (parameter). With call(), an object can use a method belonging to another object. Ex.

```
var person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + ", " + city + ", " + country;
  }
}

var person1 = {
  firstName:"John",
  lastName: "Doe"
}
person.fullName.call(person1, "Oslo", "Norway"); //Will return "John Doe, Oslo, Norway"
```

# Apply

apply() method is similar to the call(). call() method takes arguments separately. apply() method takes arguments as an array.  Ex.

```
var person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + ", " + city + ", " + country;
  }
}
var person1 = {
  firstName:"John",
  lastName: "Doe"
}
person.fullName.call(person1, ["Oslo", "Norway"]);  //Will return "John Doe, Oslo, Norway"
```

In JavaScript strict mode, if the first argument of the apply() method is not an object, it becomes the owner (object) of the invoked function. In "non-strict" mode, it becomes the global object.

# Bind

bind() returns a bound function with the context which will be invoked later. It's similar to call() & apply() but creates(for future use) a new function that, when called, has this keyword set to the provided value. Ex.

```
var person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + ", " + city + ", " + country;
  }
}
var person1 = {
  firstName:"John",
  lastName: "Doe"
}
var bound = person.fullName.bind(person1);
console.log(bound("Oslo", "Norway")); //Will return "John Doe, Oslo, Norway"
```

# Closure

JS variables can belong to the local or global scope. All functions(even nested functions) have access to the global scope & to the scope "above" them. Global variables can be made local (private) with closures using these nested functions. Ex.

```
function makeFunc() {
  var name = 'Mozilla';
  return function() {
    console.log(name);
  }
}
const myFunc = makeFunc();
myFunc(); // Mozilla
```

myFunc is a reference to the instance of the anonymous function that is created when makeFunc is run. The instance of anonymous function maintains a reference to its lexical environment, within which the variable name exists. For this reason, when myFunc is invoked, the variable name remains available for use, and "Mozilla" is passed to console.log. A closure is a function having access to the parent scope, even after the parent function has closed.

**Lexical scoping** describes how a parser resolves variable names when functions are nested. The word lexical refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope. Hence, a closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment).

# Callbacks

A callback is a function passed as an argument to another function. This technique allows a function to call another function. A callback function can run after another function has finished. Ex:

```
function greeting(name) {
 alert('Hello ' + name);
}
function processUserInput(callback) {
 const name = prompt('Please enter your name.');
 callback(name);
}
processUserInput(greeting);
```

The above example is a synchronous callback, as it is executed immediately. However, that callbacks are often used to continue code execution after an asynchronous operation has completed — these are called asynchronous callbacks. A good example is the callback functions executed inside a .then() block chained onto the end of a promise after that promise fulfills or rejects. This structure is used in many modern web APIs, such as fetch(). These async code will be covered later.

# Object Accessors

Getters and setters allow you to define functions which are object accessors. These getter & setter functions are accessed like a property, i.e. they do not need (). Ex.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "en",
  get lang() {
    return this.language;
  },
  set lang(lang) {
    this.language = lang;
  }
};
person.lang = "fr";
console.log(person.lang); // "fr"
```

# Object Constructors

Sometimes we need a "blueprint" for creating many objects of the same "type". The way to create an "object type", is to use an object constructor function. Ex.

```
function Person(first, last, age, eye) { // Name constructor functions with an upper-case first letter
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
  this.name = function() {
    return this.firstName + " " + this.lastName;
  };
}
const myFather = new Person("John", "Doe", 50, "blue");
const myMother = new Person("Sally", "Rally", 48, "green");
```

# Object Constructors(contd.)

In a constructor function this does not have a value. It is a substitute for the new object. The value of this will become the new object when a new object is created.Adding a new property to an existing object is easy. Ex:

```
myFather.nationality = "English"; // The property will be added to myFather. Not to myMother. (Not to any other person objects).
```

Adding a new method to an existing object is easy. Ex:

```
myFather.fullName = function () {  // The method will be added to myFather. Not to myMother. (Not to any other person objects).
  return this.firstName + " " + this.lastName;
};
```

You cannot add a new property to an object constructor the same way you add a new property to an existing object. Ex.

```
Person.nationality = "English";
```

# Object Constructors(contd.)

You cannot add a new method to an object constructor the same way you add a new method to an existing object. Ex.

```
Person.changeName = function (name) {
  this.lastName = name;
};
```

To add a new property or a method to a constructor, you must add it to the constructor function. Ex:

```
function Person(first, last, age, name) {
  this.firstName = first;
  this.lastName = last;
  this.nationality = "English";
  this.changeName = function (name) {
    this.lastName = name;
  };
}
```

# Built-in Constructors vs Literals

**Built-in Constructors:** JS has built-in constructors for native objects: Ex.

```
new String()    // A new String object
new Number()    // A new Number object
new Boolean()   // A new Boolean object
new Object()    // A new Object object
new Array()     // A new Array object
new RegExp()    // A new RegExp object
new Function()  // A new Function object
new Date()      // A new Date object
```

As you can see above, JavaScript has object versions of the primitive data types String, Number, and Boolean. But there is no reason to create complex objects. Primitive values(**literals**) are much faster:

- Use string literals "" or '' instead of new String().
- Use number literals 50 instead of new Number().
- Use boolean literals true / false instead of new Boolean().
- Use object literals {} instead of new Object().
- Use array literals [] instead of new Array().
- Use pattern literals /()/ instead of new RegExp().
- Use function expressions () {} instead of new Function().

# Prototype

JS objects inherit properties and methods from a prototype(can either be an object or null). Ex. Date objects inherit from Date.prototype. Array objects inherit from Array.prototype. The Object.prototype is on the top of the prototype inheritance chain. Date objects, Array objects, and Person objects inherit from Object.prototype. Usage:

- prototype property allows you to add new properties to object constructors. Ex.

```
function Person(first, last) {
  this.firstName = first;
  this.lastName = last;
}
Person.prototype.nationality = "English";
```

- prototype property also allows you to add new methods to objects constructors. Ex.

```
function Person(first, last) {
  this.firstName = first;
  this.lastName = last;
}
Person.prototype.name = function() {
  return this.firstName + " " + this.lastName;
};
```

**Note**: Only modify your own prototypes. Avoid modifying the prototypes of standard JavaScript objects.

# Prototypal Inheritance & Polyfills

When we want to read a property from an object, and it's missing, JavaScript automatically takes it from the prototype. This is called "prototypal inheritance". There can be only one [[Prototype]], which is internal & hidden. Functions Object.getPrototypeOf() / Object.setPrototypeOf() can be used to get/set the prototype. Ex:

```
const obj = {};
console.log(Object.getPrototypeOf(obj)); // Object {  }
```

If we want to find if a property is it's own or inherited, there's a built-in method available: `obj.hasOwnProperty(key)`, which returns true if obj has its own property named key.

A for..in loop will iterate over both own & inherited property but other key/value-getting methods like Object.keys, Object.values, etc ignore inherited properties.

You can add your own properties to in-built data types though this is not encouraged. Ex.

```
String.prototype.sentenceCase = function () { // this is a polyfill
  return this.charAt(0).toUpperCase() + this.substr(1).toLowerCase();
}
"hello world".sentenceCase(); // Hello world
```

# Object Metadata

Object properties, besides a value, have three special attributes (so-called “flags”):

- **writable** – if true, the value can be changed, otherwise it’s read-only. This example makes language property read-only in person object.
  Object.defineProperty(person, "language", {writable:false});
- **enumerable** – if true, then listed in loops(for..in, Object.keys), otherwise not listed. This example makes language property not enumerable in person object:
  Object.defineProperty(person, "language", {enumerable:false});
- **configurable** – if true, the property can be deleted and these attributes can be modified, otherwise not(value can be changed, if writable true). It’s a one way road. This example makes language property not configurable in person object:
  Object.defineProperty(person, "language", {configurable:false});

# Object Methods

- Object.create(proto, [descriptors]) - Creates an empty object with given proto as [[Prototype]] and optional property descriptors.
- Object.fromEntries(Array([key, value])) - Returns an object from the key, value pairs.
- Object.assign(destination_obj, source_obj1, source_obj2, ...) - Shallow copies the properties of all source objects into target. Existing property name's get overwritten.
- Object.defineProperty(object, property, descriptor) - Adding or changing a property
- Object.defineProperties(object, descriptors) - Adding or changing many properties
- Object.getOwnPropertyDescriptor(object, property) - Accessing Properties
- Object.getOwnPropertyNames(object) - Returns all properties as an array
- Object.getPrototypeOf(object) - Accessing the prototype
- Object.setPrototypeOf(object, proto) - Assigning the prototype
- Object.preventExtensions(object) - Prevents adding properties to an object
- Object.isExtensible(object) - Returns true if properties can be added to an object
- Object.seal(object) - Prevents changes of object properties (not values)
- Object.isSealed(object) - Returns true if object is sealed
- Object.freeze(object) - Prevents any changes to an object (even values)
- Object.isFrozen(object) - Returns true if object is frozen

# Class

Class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method. Use the keyword class to create a class, and always add the constructor() method. The constructor method is called each time the class object is initialized. If you do not have a constructor method, JS will add an invisible and empty constructor method. Ex.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}
const mycar = new Car("Ford");
```

**Note**: class declarations are not hoisted. Hence, you must declare a class before using it.

**Inheritance**: extends keyword can be used to create a class inheritance. A class created with a class inheritance inherits all the methods from another class. super() method refers to the parent class. By calling the super() method in the constructor method, we call the parent's constructor method and get access to the parent's properties and methods. Ex.

# Class(contd.)

```
class Model extends Car {
  constructor(brand, mod) {
    super(brand); // Calling parent class's constructor
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}
const mycar = new Model("Ford", "Mustang");
```

**Getters/Setters**

get and set keywords can be used to add getters and setters in the class. The name of the getter/setter method cannot be the same as the name of the property, in this case carname. Many programmers use an underscore character _ before the property name to separate the getter/setter from the actual property. Ex.

```
class Car {
  constructor(brand) {
    this._carname = brand;
  }
```

# Class(contd.)

```
  get carname() {
    return this._carname;
  }
  set carname(x) {
    this._carname = x;
  }
}
const mycar = new Car("Ford");
mycar.carname = "Volvo";
console.log(mycar.carname);
```

**Note**: even if the getter/setter is a method, you do not use parentheses when you want to get the property value.

Class can be considered a syntactic sugar to define a constructor together with its prototype methods. There are differences:

- First, a function created by class is labelled by a special internal property [[FunctionKind]]:"classConstructor". So it's not entirely the same as creating it manually. The language checks for that property in a variety of places. For example, unlike a regular function, it must be called with new.
- Also, a string representation of a class constructor in most JavaScript engines starts with the "class...".
- Class methods are non-enumerable. A class definition sets enumerable flag to false for all methods in the "prototype". That's good, because if we for..in over an object, we usually don't want its class methods.

# Class(contd.)

- Classes always use strict. All code inside the class is automatically in strict mode.

**Overriding a method**: By default, all methods that are not specified in child class are taken directly "as is" from parent class. But, if we specify our own method in child class, then it will be used instead. If we don't want to totally replace a parent method, but rather to build on top of it to tweak or extend its functionality, we call the parent method first before we do something in our method. Ex.

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    console.log(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    console.log(`${this.name} stands still.`);
  }
}
```

# Class(contd.)

```
class Rabbit extends Animal {
  hide() {
    console.log(`${this.name} hides!`);
  }
  stop() {
    super.stop(); // call parent stop
    this.hide(); // and then hide
  }
}
const rabbit = new Rabbit("White Rabbit");
rabbit.run(5); // White Rabbit runs with speed 5
rabbit.stop(); // White Rabbit stands still. White rabbit hides!
```

**Static properties & methods** are defined on the class itself, and not on the prototype. That means you cannot call a static property or method on the object (mycar), but on the class (Car). These properties and methods are common across all the instances of the class. This means that they do not belong to the instance but to the class only. Ex.

```
class Car {
  constructor(brand) {
    this.carname = brand;
```

# Class(contd.)

```
  }
  static count = 0;
  static hello() {
    return "Hello!!";
  }
  present() {
    return 'I have a ' + this.carname;
  }
}
const mycar = new Car("Maruti");
console.log(Car.count); // valid & logs 0
console.log(mycar.count); // this would raise an error.
console.log(Car.hello()); // valid
console.log(mycar.hello()); // this would raise an error.
```

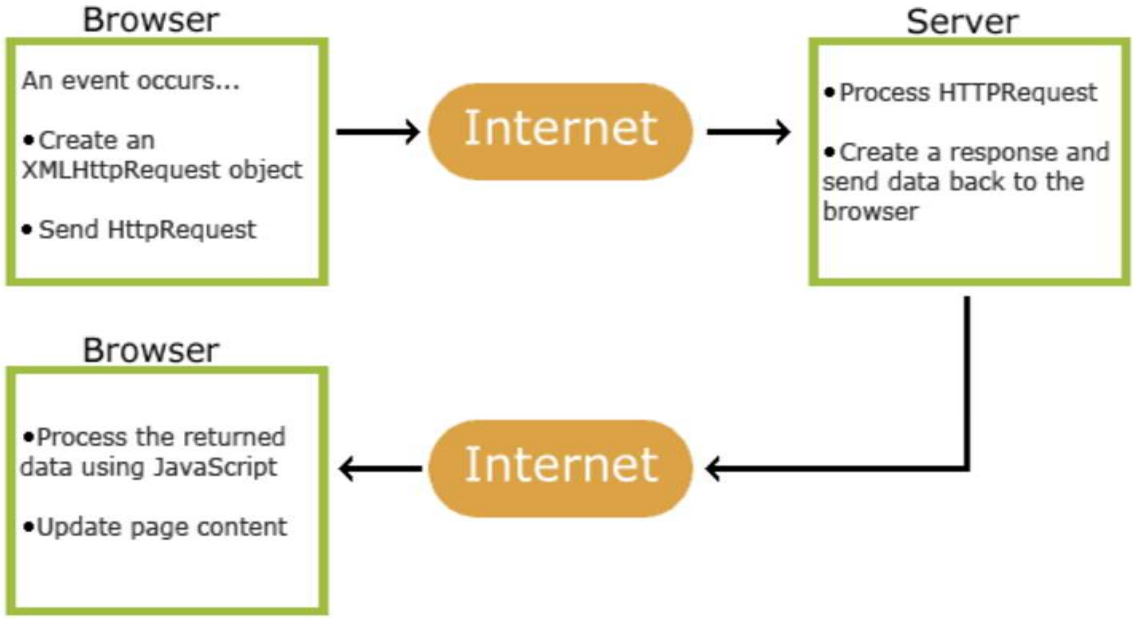
**Note**: Static properties and methods are also inherited in every class except built in classes.

# AJAX

AJAX = Asynchronous JavaScript And XML. AJAX just uses a combination of:

- A browser built-in XMLHttpRequest object (to request data from a web server)
- JavaScript and HTML DOM (to display or use the data)

AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page. How AJAX works:



# AJAX(contd.)

The XMLHttpRequest object can be used to exchange data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

Steps:

- Create an XMLHttpRequest object. Ex.

  ```
  const xhttp = new XMLHttpRequest();
  ```

- A callback function is a function passed as a parameter to another function. In this case, the callback function should contain the code to execute when the response is ready. Ex:

  ```
  xhttp.onload = function() {
    // What to do when the response is ready
  }
  ```

- To send a request to a server, you can use the open() and send() methods of the XMLHttpRequest object. Ex:

  ```
  xhttp.open("GET", "ajax_info.txt");
  xhttp.send();
  ```

**XMLHttpRequest Object Methods**

# AJAX(contd.)

- new XMLHttpRequest(): Creates a new XMLHttpRequest object
- abort(): Cancels the current request
- getResponseHeader(): Returns specific header information
- getAllResponseHeaders(): Returns all the header information from the server resource
- open(method, url, async, user, psw): Specifies the request method: the request type GET or POST. Url is the file location. Async is a boolean flag where true (asynchronous) or false (synchronous). User is an optional user name. Psw is an optional password.
- send(): Sends the request to the server(Used for GET requests)
- send(string): Sends the request to the server(Used for POST requests)
- setRequestHeader(): Adds a label/value pair to the header to be sent

**XMLHttpRequest Object Properties**

- onload: Defines a function to be called when the request is received (loaded)
- onreadystatechange: Defines a function to be called when the readyState property changes
- readyState: Holds the status of the XMLHttpRequest. 0: request not initialized, 1: server connection

# AJAX(contd.)

established, 2: request received, 3: processing request 4: request finished and response is ready

- responseText: Returns the response data as a string
- responseXML: Returns the response data as XML data status.
- status: Returns the status-number of a request. Ex. 200: "OK", 403: "Forbidden", etc.
- statusText: Returns the status-text (e.g. "OK" or "Not Found")

Example:

```
const xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    console.log(this.responseText);
  }
};
xhttp.open("GET", "ajax_info.txt");
xhttp.send();
```

**Note**: Modern Browsers can use Fetch API instead of the XMLHttpRequest Object.

# Browser API

API stands for Application Programming Interface. A Browser API can extend the functionality of a web browser. All browsers have a set of built-in Web APIs to support complex operations, and to help accessing data.

**Forms Validation API**

**Constraint Validation DOM Methods:**

- checkValidity(): Returns true if an input element contains valid data.
- setCustomValidity(): Sets the validationMessage property of an input element.

**Constraint Validation DOM Properties:**

- validity: Contains boolean properties related to the validity of an input element.
  - customError: Set to true, if a custom validity message is set.
  - patternMismatch: Set to true, if an element's value does not match its pattern attribute.
  - rangeOverflow: Set to true, if an element's value is greater than its max attribute.
  - rangeUnderflow: Set to true, if an element's value is less than its min attribute.
  - stepMismatch: Set to true, if an element's value is invalid per its step attribute.
  - tooLong: Set to true, if an element's value exceeds its maxLength attribute.
  - typeMismatch: Set to true, if an element's value is invalid per its type attribute.
  - valueMissing: Set to true, if an element (with a required attribute) has no value.
  - valid: Set to true, if an element's value is valid.

# Browser API(contd.)

- validationMessage: Contains the message a browser will display when the validity is false.
- willValidate: Indicates if an input element will be validated.

Example:

```
<input id="id1" type="number" min="100" max="300" required onchange="myFunction()">
<script>
function myFunction() {
  const inpObj = document.getElementById("id1");
  if (!inpObj.checkValidity()) {
    console.log(inpObj.validationMessage); // For input less than 100 -> Value must be greater than or equal to 100. For input greater than 300 -> Value must be less than or equal to 300.
  }
}
</script>
```

**History API**

The Web History API provides easy methods to access the windows.history object. The window.history object contains the URLs (Web Sites) visited by the user.

# Browser API(contd.)

**History Object Properties:**

- length: Returns the number of URLs in the history list. Ex. window.history.length;

**History Object Methods:**

- back(): Loads the previous URL in the history list. Ex. window.history.back();
- forward(): Loads the next URL in the history list. Ex. window.history.forward();
- go(): Loads a specific URL from the history list. Ex.

  window.history.go(-2); // 2 pages back

  window.history.go(2); // 2 pages forward

**Storage API**

Storage API is a simple syntax for storing and retrieving data in the browser. There are two storage objects:

1. **Local Storage**: The localStorage object provides access to a local storage for a particular Web Site. It allows you to store, read, add, modify, and delete data items for that domain. The data is stored with no expiration date, and will not be deleted when the browser is closed. The data will be available for days, weeks, and years. It's methods & properties:
   - length: Returns the number of data items stored in the Storage object. Ex. localStorage.length;
   - key(n): Returns the name of the nth key in the storage. Ex. localStorage.key(3);
   - getItem(keyname): Returns the value of the specified key name. Ex. localStorage.getItem("name");
   - removeItem(keyname): Removes that key from the storage. Ex. localStorage.removeItem("name");

# Browser API(contd.)

- setItem(keyname, value): Adds that key to the storage, or update that key's value if it already exists. Ex. localStorage.setItem("name", "John Doe");
- clear(): Empty all key out of the storage. Ex. localStorage.clear();

1. **Session Storage**: The sessionStorage object is identical to the localStorage object. The difference is that the sessionStorage object stores data for one session. The data is deleted when the browser/tab is closed. It's methods & properties:
   - length: Returns the number of data items stored in the Storage object. Ex. sessionStorage.length;
   - key(n): Returns the name of the nth key in the storage. Ex. sessionStorage.key(3);
   - getItem(keyname): Returns the value of the specified key name. Ex. sessionStorage.getItem("name");
   - removeItem(keyname): Removes that key from the storage. Ex. sessionStorage.removeItem("name");
   - setItem(keyname, value): Adds that key to the storage, or update that key's value if it already exists. Ex. sessionStorage.setItem("name", "John Doe");
   - clear(): Empty all key out of the storage. Ex. sessionStorage.clear();

**Fetch API**

Fetch API interface allows web browser to make HTTP requests to web servers. Ex:

```
fetch ("fetch_info.txt")
.then(x => x.text())
.then(y => console.log(y));
```

# Browser API(contd.)

**Geolocation API**

Geolocation API is used to get the geographical position of a user. Since this can compromise privacy, the position is not available unless the user approves it. The getCurrentPosition() method is used to return the user's position. The second parameter of the getCurrentPosition() method is used to handle errors. It specifies a function to run if it fails to get the user's location. Ex.

```
function getLocation() {
  Navigator.geolocation ? navigator.geolocation.getCurrentPosition(showPosition, showError) : console.log("Geolocation is not supported by this browser.");
}
function showPosition(position) {
  console.log(`Latitude: ${position.coords.latitude} Longitude: ${position.coords.longitude}`);
}
function showError(error) {
  console.log("An error occurred");
}
```

The Geolocation object also has other interesting methods:

- watchPosition() - Returns the current position of the user and continues to return updated position as the user moves (like the GPS in a car).
- clearWatch() - Stops the watchPosition() method.

# Window Object

Browser Object Model (BOM) allows JavaScript to "talk to" the browser. Commonly known as the Window Object, it represents the browser's window. All global JavaScript objects, functions, and variables automatically become members of the window object. Global variables & functions are properties of the window object. Even the document object (of the HTML DOM) is a property of the window object. Ex. window.document.getElementById("header"); is the same as: document.getElementById("header");

It has the following properties & methods:

- **screen** object contains information about the user's screen. Properties:
  - screen.width - returns the width of the visitor's screen in pixels.
  - screen.height - returns the height of the visitor's screen in pixels.
  - screen.availWidth - returns the width of the visitor's screen, in pixels, minus the Windows Taskbar.
  - screen.availHeight - returns the height of the visitor's screen, in pixels, minus Windows Taskbar.
- **location** object can be used to get the current page address (URL) and to redirect the browser to a new page. Properties:
  - location.href - returns the href (URL) of the current page
  - location.hostname - returns the domain name of the web host
  - location.pathname - returns the path and filename of the current page
  - location.protocol - returns the web protocol used (http: or https:)
  - location.assign() - loads a new document

# Window Object(contd.)

- **Popup Boxes:** JS has three kind of popup boxes: Alert box, Confirm box, and Prompt box.
  - **Alert box** is often used if you want to make sure information comes through to the user. When an alert box pops up, the user will have to click "OK" to proceed. Syntax: window.alert("*sometext*"); Ex:

    ```
    alert("I am an alert box!");
    ```
  - **Confirm box** is often used if you want the user to verify or accept something. When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed. If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false. Syntax: window.confirm("sometext"); Ex:

    ```
    if (confirm("Press a button!")) {
      console.log("You pressed OK!");
    } else {
      console.log("You pressed Cancel!");
    }
    ```
  - **Prompt box** is often used if you want the user to input a value before entering a page. When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value. If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null. Syntax: window.prompt("sometext","defaultText"); Ex:

    ```
    let person = prompt("Please enter your name", "Harry Potter");
    person == null || person == "" ? console.log("User cancelled the prompt.") : console.log(`Hello ${person}! How are you today?`);
    ```

# Window Object(contd.)

- **Timing Events:** JS The window object allows execution of code at specified time intervals. These time intervals are called timing events. The two key methods to use with JavaScript are:
  - setTimeout Executes a function, after waiting a specified delay of milliseconds. Syntax: setTimeout(function, delay). The first parameter is a function to be executed. The second parameter indicates the number of milliseconds before execution. Ex:

    ```
    setTimeout(function() {
      alert('Hello'); // alerts Hello after 3 seconds
    }, 3000)
    ```

    The clearTimeout() method stops the execution of the function specified in setTimeout(). The clearTimeout() method uses the variable returned from setTimeout().

    ```
    const timeout = setTimeout(function, milliseconds);
    clearTimeout(timeout);
    ```
  - Same as setTimeout(), but repeats the execution of the function continuously. Syntax: setInterval(function, delay). The first parameter is a function to be executed. The second parameter indicates the number of milliseconds before execution. Ex:

    ```
    setInterval(function() {
      alert('Hello'); // alerts Hello continuously after every 3 seconds
    }, 3000)
    ```

    The clearInterval() method stops the execution of the function specified in setInterval(). The clearInterval() method uses the variable returned from setInterval().

# Window Object(contd.)

```
const interval = setInterval(function, milliseconds);
clearInterval(interval);
```

- **Cookies**: Cookies let you store user information in web pages. Cookies are saved in name-value pairs like: username = John Doe. When a browser requests a web page from a server, cookies belonging to the page are added to the request. This way the server gets the necessary data to "remember" information about users. JS can create, read, and delete cookies with the document.cookie property. With JavaScript, a cookie can be created like this: `document.cookie = "username=John Doe";`
  You can also add an expiry date (in UTC time). By default, the cookie is deleted when the browser is closed:
  `document.cookie = "username=John Doe; expires=Thu, 18 Dec 2022 12:00:00 UTC";`
  With a path parameter, you can tell the browser what path the cookie belongs to. By default, it belongs to the current page.
  `document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";`
  cookies can be read like this: `let x = document.cookie;` document.cookie will return all cookies in one string much like:
  `cookie1=value; cookie2=value; cookie3=value;`
  We can change a cookie the same way as you create it:
  `document.cookie = "username=John Smith; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";`
  Deleting a cookie is very simple. Just set the expires parameter to a past date. You don't have to specify a cookie value when you delete a cookie: `document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";`
  If you set a new cookie, older cookies are not overwritten. The new cookie is added to document.cookie, so if you read document.cookie again you will get something like: `cookie1 = value; cookie2 = value;`

# Async JS

Functions running in parallel with other functions are called asynchronous. Example: setTimeout(). Another could be XMLHttpRequest.

```
function myDisplayer(some) {
  console.log(some);
}
function getFile(myCallback) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
                  myCallback(this.responseText);
                  } else {
                  myCallback("Error: " + req.status);
                  }
  }
  req.send();
}
getFile(myDisplayer);
```

# Promises

A Promise is a JavaScript object that links producing code and consuming code. "Producing code" is code that can take some time. "Consuming code" is code that must wait for the result. JS Promise object contains both the producing code and calls to the consuming code. Syntax:

```
let promise = new Promise(function(resolve, reject) {
// "Producing Code" (May take some time)
  resolve(result); // when successful
  reject(result);  // when error
});
promise.then( // "Consuming Code" (Must wait for a fulfilled Promise)
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

When the executing code obtains the result, it should call one of the two callbacks. When promise is success, then resolve(result value) runs but if promises fails/errors then reject(error object) runs.

A JavaScript Promise object can be: Pending, Fulfilled or Rejected.

The Promise object supports two properties: state & result. While a Promise object's state is "pending" (working), the result is undefined.it's "fulfilled", the result is a value & it's "rejected", the result is an error object. These promise properties are not manually accessible.

# Async/Await

Async and Await make promises easier to write. Async makes a function return a Promise. Await makes a function wait for a Promise.

The keyword **async** before a function makes the function return a promise:

```
function myFunction() {
  return Promise.resolve("Hello");
}
```

is same as

```
async function myFunction() {
  return "Hello";
}
```

```
myFunction().then(
  function(value) { /* code if successful */ }
);
```

is same as

```
myFunction().then(
function(value) {console.log(value);}
);
```

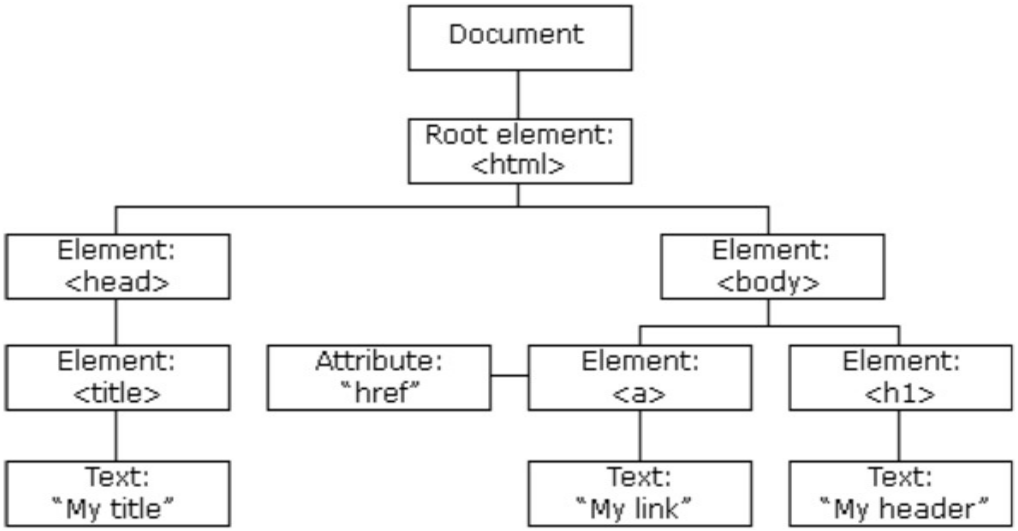
The keyword **await** before a function makes the function wait for a promise & returns a value. Syntax: `let value = await promise;`

The await keyword can only be used inside an async function. Ex.

```
async function myDisplay() {
  const promise = new Promise(function(resolve, reject) {
    resolve("I love You !!");
  });
```

# DOM Introduction

When a web page is loaded, the browser creates a Document Object Model(DOM) of the page. With this HTML DOM, JavaScript can access and change all the elements of an HTML document. The HTML DOM model is constructed as a tree of Objects:




DOM is a standard object model and programming interface for HTML. It defines:

- The HTML elements as **objects**
- The **properties** of all HTML elements
- The **methods** to access all HTML elements
- The **events** for all HTML elements

# DOM Introduction(contd.)

The HTML DOM can be accessed with JavaScript. In the DOM, all HTML elements are defined as objects. The programming interface is the properties and methods of each object.

- A property is a value that you can get or set (like changing the content of an HTML element).
- A method is an action you can do (like add or deleting an HTML element).

Example:

```
<html>
 <body>
  <p id="demo"></p>
  <script>
   document.getElementById("demo").innerHTML = "Hello World!";
  </script>
 </body>
</html>
```

Here, getElementById is a method, while innerHTML is a property.

The document object represents your web page. If you want to access any element in an HTML page, you always start with accessing the document object.

# DOM Elements

With JavaScript when we want to access/manipulate HTML elements we need to find the elements first. Methods:

- **Finding HTML elements by id**: If the element is found, the method will return the element as an object. If the element is not found, it will return null. Ex.
  `const elem = document.getElementById("intro");`
- **Finding HTML elements by tag name**: If the element(s) is/are found, the method will return an array-like HTMLCollection of objects. If the element is not found, it will return an empty HTMLCollection. Ex:
  `const elements = document.getElementsByTagName("p");`
- **Finding HTML elements by class name**: If the element(s) is/are found, the method will return an array-like HTMLCollection of objects. If the element is not found, it will return an empty HTMLCollection. Ex.
  `const x = document.getElementsByClassName("intro");`
- **Finding HTML elements by CSS selectors**: To find HTML elements that match a specified CSS selector (id, class names, types, attributes, values of attributes, etc), use the querySelectorAll() method. If the element(s) is/are found, the method will return an array-like Node List. If the element is not found, it will return an empty Node List. Ex.
  `const x = document.querySelectorAll("p.intro");`
- **Finding HTML elements by HTML object collections**: To find all elements of one type like forms, images, anchors, etc. Types:
  - document.anchors
  - document.body
  - document.embeds

# DOM Elements

- document.forms
- document.head
- document.images
- document.links
- document.scripts
- document.title

Example:

```
<form id="frm1">
  Name: <input type="text" name="name" value="Donald Duck"><br>
  <input type="submit" value="Submit">
</form>
<p id="demo"></p>
<script>
const x = document.forms["frm1"];
let text = "";
for (let i = 0; i < x.length ;i++) {  text += x.elements[i].value + "<br>";  }
document.getElementById("demo").innerHTML = text;
</script>
```

# DOM HTML

To change HTML content of an element, use the innerHTML property. Syntax: document.getElementById(id).innerHTML = new HTML. Ex:

```
<p id="p1">Hello World!</p>
<script>
 document.getElementById("p1").innerHTML = "New <b>text!</b>"; // changes the HTML content of a <p> element
</script>
```

To change text content of an element, use the innerText property. Syntax: document.getElementById(id).innerText = text. Ex:

```
<p id="p1">Hello World!</p>
<script>
 document.getElementById("p1").innerText = "New text!"; // changes the text content of a <p> element
</script>
```

To change the value of an HTML attribute, use this syntax: document.getElementById(id).attribute = new value; Ex:

```
<img id="myImage" src="smiley.gif">
<script>
 document.getElementById("myImage").src = "landscape.jpg"; // changes the value of the src attribute of an <img> element
</script>
```

In JavaScript, document.write() can be used to write directly to the HTML output stream. Never use document.write() after the document is loaded. It will overwrite the document.

# DOM Forms

HTML form validation can be done by JavaScript. If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted. Ex.

```
<form name="myForm" onsubmit="return validateForm()" method="post">
  Name: <input type="text" name="fname">
  <input type="submit" value="Submit">
</form>
<script>
function validateForm() {
  if (document.forms["myForm"]["fname"].value === "") {
    alert("Name must be filled out");
    return false; // false is returned to stop onsubmit
  }
}
</script>
```

# DOM CSS

To change the style of an HTML element, use this syntax:

document.getElementById(id).style.property = new style; Ex:

```
<p id="p2">Hello World!</p>
<script>
document.getElementById("p2").style.color = "blue"; // changes the style of a <p> element
</script>
```

**Note**: For css properties which have 2 words e.g. background-color, padding-top, etc. should be written in camelcase e.g. backgroundColor, paddingTop. Etc.

```
<p id="p2">Hello World!</p>
<script>
document.getElementById("p2").style.backgroundColor = "blue"; // changes the style of a <p> element
</script>
```

# DOM Events

A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element. To execute code when a user clicks on an element, add JavaScript code to an HTML event attribute: onclick=JavaScript. Ex.

```
<h1 onclick="changeText(this)">Click on this text!</h1> <!-- this here represents the h1 element -->
<script>
function changeText(elem) {
  elem.innerHTML = "Ooops!";
}
</script>
```

The HTML DOM allows you to assign events to HTML elements using JS: Example Assign an onclick event to a button element. Ex:

```
<button>Click here</button>
<script>
  document.getElementById("myBtn").onclick = displayDate;
</script>
```

**Hover Event:** The onmouseover and onmouseout events can be used to trigger a function when the user mouses over, or out of, an HTML element.

**Click Event**: The onmousedown, onmouseup, and onclick events are all parts of a mouse-click. First when a mouse-button is clicked, the onmousedown event is triggered, then, when the mouse-button is released, the onmouseup event is triggered, finally, when the mouse-click is completed, the onclick event is triggered.

# DOM Events(contd.)

**Keyboard Event**: The keydown, keyup, and keypress events are all parts of a keyboard press. First when a keyboard-button is tapped, the keydown event is triggered, then, when the keyboard-button is released, the keyup event is triggered, finally, when the keyboard press is completed, the keypress event is triggered.

**addEventListener()**: This method attaches an event handler to the specified element without overwriting existing event handlers.
Syntax: element.addEventListener(event, function, useCapture);

- The first parameter is the type of the event (like "click" or "mousedown" or any other HTML DOM Event.)
- The second parameter is the function we want to call when the event occurs.
- The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

**Event Bubbling or Event Capturing:** There are two ways of event propagation in the HTML DOM, bubbling and capturing.
Event propagation is a way of defining the element order when an event occurs. If you have a <p> element inside a  element, and the user clicks on the <p> element, which element's "click" event should be handled first?
In bubbling the inner most element's event is handled first and then the outer: the <p> element's click event is handled first, then the  element's click event.
In capturing the outer most element's event is handled first and then the inner: the  element's click event will be handled first, then the <p> element's click event.
With the addEventListener() method you can specify the propagation type by using the "useCapture" parameter. The default value is false, which will use the bubbling propagation, when the value is set to true, the event uses the capturing propagation.

# DOM Events(contd.)

Example: `document.getElementById("myBtn").addEventListener("click", displayDate);` Note that you don't use the "on" prefix for the event; use "click" instead of "onclick".

You can add many event handlers to one element. Ex.

```
element.addEventListener("mouseover", myFunction);
element.addEventListener("click", mySecondFunction);
element.addEventListener("mouseout", myThirdFunction);
```

You can add many event handlers of the same type to one element, i.e two "click" events. Ex.

```
element.addEventListener("click", myFunction);
element.addEventListener("click", mySecondFunction);
```

You can add event listeners to any DOM object not only HTML elements. i.e the window object. Ex:

```
window.addEventListener("resize", function(){
  document.getElementById("demo").innerHTML = sometext;
});
```

You can easily remove an event listener by using the removeEventListener() method. Ex:

```
element.removeEventListener("mousemove", myFunction);
```

When using the addEventListener() method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup.

# DOM Navigation

With the HTML DOM, you can navigate the node tree using node relationships. The nodes in the node tree have a hierarchical relationship to each other. The terms parent, child, and sibling are used to describe the relationships.

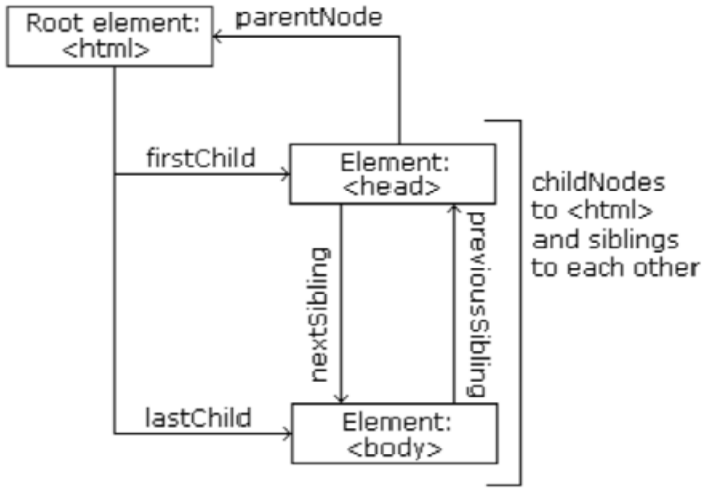
- In a node tree, the top node is called the root (or root node)
- Every node has exactly one parent, except the root (which has no parent)
- A node can have a number of children
- Siblings (brothers or sisters) are nodes with the same parent

```
<html>

  <head>
    <title>DOM Tutorial</title>
  </head>

  <body>
    <h1>DOM Lesson one</h1>
    <p>Hello world!</p>
  </body>

</html>
```



# DOM Navigation(contd.)

From the HTML before you can read:

- <html> is the root node
- <html> has no parents
- <html> is the parent of <head> and <body>
- <head> is the first child of <html>
- <body> is the last child of <html>

and:

- <head> has one child: <title>
- <title> has one child (a text node): "DOM Tutorial"
- <body> has two children: <h1> and <p>
- <h1> has one child: "DOM Lesson one"
- <p> has one child: "Hello world!"
- <h1> and <p> are siblings

We can use the following node properties to navigate between nodes with JavaScript:

- parentNode. Ex: console.log(document.body.parentNode); // <html> element
- childNodes[nodenumber]. Ex: console.log(document.head.childNodes(0)); // <title> element
- firstChild. Ex: console.log(document.body.firstChild); // <h1> element

# DOM Navigation(contd.)

- lastChild. Ex: console.log(document.body.lastChild); // <p> element
- nextSibling. Ex: console.log(document.body.firstChild.nextSibling); // <p> element
- previousSibling. Ex: console.log(document.body.lastChild.previousSibling); // <h1> element

**nodeName** property specifies the name of a node. It always contains the uppercase tag name of an HTML element. Features:

- nodeName is read-only
- nodeName of an element node is the same as the tag name
- nodeName of an attribute node is the attribute name
- nodeName of a text node is always #text
- nodeName of the document node is always #document

Example

```
<h1 id="id01">My First Page</h1>
<p id="id02"></p>
<script>
  document.getElementById("id02").innerHTML = document.getElementById("id01").nodeName; // H1
</script>
```

# DOM Navigation(contd.)

**nodeValue** property specifies the value of a node. Features:

- nodeValue for element nodes is null
- nodeValue for text nodes is the text itself
- nodeValue for attribute nodes is the attribute value

Example:

```
<h1 id="id01">My First Page</h1>
<p id="id02"></p>
<p id="id03"></p>
<script>
  document.getElementById("id02").innerHTML = document.getElementById("id01").firstChild.nodeValue; // My First Page
  document.getElementById("id03").innerHTML = document.getElementById("id01").childNodes[0].nodeValue; // My First Page
</script>
```

# DOM Nodes

To add a new element to the HTML DOM, you must create the element node first, and then append it to an existing element.

Example:

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>
<script>
  const para = document.createElement("p"); // creates a new <p> element
  para.innerText = "This is new.";
  document.getElementById("div1").appendChild(para); // appends the new element to the existing element
</script>
```

The appendChild() method in the previous example, appended the new element as the last child of the parent. If you don't want that you can use the insertBefore() method to insert as a sibling before another element. Ex:

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>
<script>
  const para = document.createElement("p");
```

# DOM Nodes(contd.)

```
  para.innerText = "This is new.";
  document.getElementById("div1").insertBefore(para, document.getElementById("p1"));
</script>
```

To remove an HTML element, use the remove() method. Example:

```
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
<script>
  document.getElementById("p1").remove();
</script>
```

You can also remove a child node. For that you have to find the parent node to remove an element. Example:

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
</div>
<script>
  const parent = document.getElementById("div1");
  const child = document.getElementById("p1");
  parent.removeChild(child);
</script>
```

# DOM Nodes(contd.)

To replace an element to the HTML DOM, use the replaceChild() method. Ex:

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>
<script>
  const para = document.createElement("p");
  para.innerText = "This is new.";
  const parent = document.getElementById("div1");
  const child = document.getElementById("p1");
  parent.replaceChild(para, child);
</script>
```

# HTML Collection & HTML NodeList

**HTML Collection**

To The getElementsByTagName() method returns an HTMLCollection object. An HTMLCollection object is an array-like list (collection) of HTML elements. It has a length property that defines the number of elements in an HTMLCollection. The length property is useful when you want to loop through the elements in a collection. Example:

```
const myCollection = document.getElementsByTagName("p");
for (let i = 0; i < myCollection.length; i++) {
  myCollection[i].style.color = "red"; // Changes the text color of all <p> elements:
}
```

An HTMLCollection is NOT an array! It may look like an array, but it is not. You can loop through the list and refer to the elements with a number (just like an array). However, you cannot use array methods like valueOf(), pop(), push(), or join() on it.

**NodeList**

A NodeList object is a list (collection) of nodes extracted from a document. A NodeList object is almost same as an HTMLCollection object. Some (older) browsers return a NodeList object instead of an HTMLCollection for methods like getElementsByClassName(). All browsers return a NodeList object for the property childNodes. Browsers return a NodeList object for method querySelectorAll(). The length property is useful when you want to loop through the elements in a node list. Example:

```
const nodeList = document.querySelectorAll("p");
for (let i = 0; i < nodeList.length; i++) {
```

# HTML Collection & HTML NodeList(contd.)

```
  myCollection[i].style.color = "red"; // Changes the text color of all <p> elements:
}
```

An Node List is NOT an array! It may look like an array, but it is not. You can loop through the list and refer to the elements with a number (just like an array). However, you cannot use array methods like valueOf(), pop(), push(), or join() on it.

**Difference Between an HTMLCollection and a NodeList**

- An HTMLCollection is a collection of HTML elements. A NodeList is a collection of document nodes.
- HTMLCollection items can be accessed by their name, id, or index number. NodeList items can only be accessed by their index number.
- Only the NodeList object can contain attribute nodes and text nodes.

**Similarities Between an HTMLCollection and a NodeList**

- Both an HTMLCollection object and a NodeList object is an array-like list (collection) of objects.
- Both have a length property defining the number of items in the list (collection).
- Both provide an index (0, 1, 2, 3, 4, ...) to access each item like an array.

# Thank You