# JavaScript

**JS Async: -**

**JavaScript 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.

**Sequence Control: -**

Sometimes you would like to have better control over when to execute a function.

Suppose you want to do a calculation and then display the result.

You could call a calculator function (myCalculator), save the result, and then call another function (myDisplayer) to display the result.

Or you could call a calculator function (myCalculator), and let the calculator function call the display function (myDisplayer)

The problem with the first example above, is that you have to call two functions to display the result.

The problem with the second example, is that you cannot prevent the calculator function from displaying the result.

Using a **callback**, you could call the calculator function (myCalculator) with a callback (myCallback), and let the calculator function run the callback after the calculation is finished.

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}
myCalculator(5, 5, myDisplayer);
```

In the example above, myDisplayer is a called a callback function.

It is passed to myCalculator() as an argument.

When you pass a function as an argument, remember not to use parenthesis.

Callbacks really shine in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

**Asynchronous JavaScript: -**

Functions running in parallel with other functions are called asynchronous.

A good example is JavaScript setTimeout().

```
setTimeout(myFunction, 3000);
function myFunction() {
  document.getElementById("demo").innerHTML = "I love You !!";
}
```

myFunction is passed to setTimeout() as an argument.  myFunction is used as a callback

Instead of passing the name of a function as an argument to another function, you can always pass a whole function instead.

```
setTimeout(function() { myFunction(" Hello "); }, 3000);
function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
}
```

function(){ myFunction(" Hello "); } is used as a callback. It is a complete function. The complete function is passed to setTimeout() as an argument.

**JavaScript Promises: -**

A Promise contains both the producing code and calls to the consuming code.

Producing code is code that can take some time.

Consuming code is code that must wait for the result.

A Promise is an Object that links Producing code and Consuming code.

**Promise Syntax: -**

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

**Promise Object Properties: -**

| **myPromise.state** | **myPromise.result** |
|---|---|
| pending | undefined |
| fulfilled | a result value |
| rejected | an error object |

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
let myPromise = new Promise(function(myResolve, myReject) {
  let x = 0;
// The producing code (this may take some time)
  if (x == 0) {
    myResolve("OK");
  } else {
    myReject("Error");
  }
});
myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
```

**Using Promise for a Timeout: -**

```
let myPromise = new Promise(function(myResolve, myReject) {
  setTimeout(function() { myResolve(" Hello "); }, 3000);
});
myPromise.then(function(value) {
  document.getElementById("demo").innerHTML = value;
});
```

**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.

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

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

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

The await keyword makes the function pause the execution and wait for a resolved promise before it continues.

```
let value = await promise;
```

Let’s take an example.

```
async function myDisplay() {

  let myPromise = new Promise(function(resolve, reject) {

    resolve(" Hello ");

  });

  document.getElementById("demo").innerHTML = await myPromise;

}

myDisplay();
```

The two arguments (resolve and reject) are pre-defined by JavaScript.

**Waiting for a Timeout: -**

```
async function myDisplay() {

  let myPromise = new Promise(function(resolve) {

    setTimeout(function() {resolve(" Hello ");}, 3000);

  });

  document.getElementById("demo").innerHTML = await myPromise;

}

myDisplay();
```

**Web APIs: -**

API stands for Application Programming Interface.

It can extend the functionality of the browser. It can greatly simplify complex functions and can provide easy syntax to complex code.

A Web API is an application programming interface for the Web.

A Browser API can extend the functionality of a web browser.

A Server API can extend the functionality of a web server.

**Browser APIs: -**

All browsers have a set of built-in Web APIs to support complex operations, and to help accessing data.

For example, the Geolocation API can return the coordinates of where the browser is located.

Get the latitude and longitude of the user's position.

```
const myElement = document.getElementById("demo");
function getLocation() {
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(showPosition);
  } else {
    myElement.innerHTML = "Geolocation is not supported by this browser.";
  }
}
function showPosition(position) {
  myElement.innerHTML = "Latitude: " + position.coords.latitude +
  "<br>Longitude: " + position.coords.longitude;
}
```

**Third Party APIs: -**

Third party APIs are not built into your browser.

To use these APIs, you will have to download the code from the Web.

Ex: - YouTube API - Allows you to display videos on a web site.

**JavaScript Validation API: -**

**Constraint Validation DOM Methods: -**

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

If an input field contains invalid data, display a message. Returns true if an input element contains valid data.

```
<input id="id1" type="number" min="100" max="300" required>

<button onclick="myFunction()">OK</button>

<p id="demo"></p>

<script>

function myFunction() {

  const inpObj = document.getElementById("id1");

  if (!inpObj.checkValidity()) {

    document.getElementById("demo").innerHTML = inpObj.validationMessage;

  }

}

</script>
```

| Property | Description |
|---|---|
| validity | Contains boolean properties related to the validity of an input element. |
| validationMessage | Contains the message a browser will display when the validity is false. |
| willValidate | Indicates if an input element will be validated. |

**Validity Properties: -**

The validity property of an input element contains a number of properties related to the validity of data.

| Property | Description |
|---|---|
| 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. |

If the number in an input field is greater than 100 (the input's max attribute), display a message.

```
<input id="id1" type="number" max="100">

<button onclick="myFunction()">OK</button>

<p id="demo"></p>

<script>

function myFunction() {

  let text = "Value OK";

  if (document.getElementById("id1").validity.rangeOverflow) {

    text = "Value too large";

  }

}

</script>
```

**Web 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.

**The History back() Method: -**

The back() method loads the previous URL in the windows.history list.

**The History go() Method: -**

The go() method loads a specific URL from the history list.

```
button onclick="myFunction()">Go Back 2 Pages</button>

<script>

function myFunction() {

  window.history.go(-2);
}
</script>
```

**History Object Property: -**

**history.length -** Returns the number of URLs in the history list.

**Web Storage API: -**

The Web Storage API is a simple syntax for storing and retrieving data in the browser.

```
localStorage.setItem("name", "John Doe");
localStorage.getItem("name"); // John Doe
```

**The localStorage Object: -**

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 sessionStorage Object: -**

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 is closed.

```
sessionStorage.setItem("name", "John Doe");

sessionStorage.getItem("name");
```

**Storage Object Properties and Methods: -**

| **Property/Method** | **Description** |
|---|---|
| key(n) | Returns the name of the nth key in the storage |
| length | Returns the number of data items stored in the Storage object |
| getItem(keyname) | Returns the value of the specified key name |
| setItem(keyname, value) | Adds a key to the storage, or updates a key value (if it already exists) |
| removeItem(keyname) | Removes that key from the storage |
| clear() | Empty all key out of the storage |

**Web Workers API: -**

When executing scripts in an HTML page, the page becomes unresponsive until the script is finished.

A web worker is a JavaScript running in the background, without affecting the performance of the page.

You can continue to do whatever you want: clicking, selecting things, etc., while the web worker runs in the background.

**Create a Web Worker File: -**

Now, let's create our web worker in an external JavaScript.

Here, we create a script that counts. The script is stored in the "demo_workers.js" file.

```
let i = 0;

function timedCount() {

  i ++;

  postMessage(i);

  setTimeout("timedCount()",500);

}

timedCount();
```

postMessage() method - which is used to post a message back to the HTML page.

**Create a Web Worker Object: -**

we need to call it from an HTML page.

The following lines checks if the worker already exists, if not - it creates a new web worker object and runs the code in "demo_workers.js"

```
if (typeof(w) == "undefined") {
  w = new Worker("demo_workers.js");
}
```

Then we can send and receive messages from the web worker.

```
w.onmessage = function(event){
  document.getElementById("result").innerHTML = event.data;
};
```

When the web worker posts a message, the code within the event listener is executed. The data from the web worker is stored in event.data.

**Terminate a Web Worker: -**

it will continue to listen for messages (even after the external script is finished) until it is terminated.

To terminate a web worker, and free browser/computer resources, use the terminate() method.

```
w.terminate();
```

**Reuse the Web Worker: -**

```
w = undefined;
```

**JavaScript Fetch API: -**

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

The example below fetches a file and displays the content.

```
fetch(file)
.then(x => x.text())
.then(y => myDisplay(y));
```

Fetch is based on async and await, the example above might be easier to understand like this.

```
async function getText(file) {
  let x = await fetch(file);
  let y = await x.text();
  myDisplay(y);
}
```

**Web Geolocation API: -**

The HTML 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 Geolocation API will only work on secure contexts such as HTTPS.

```
<script>

const x = document.getElementById("demo");

function getLocation() {

  if (navigator.geolocation) {

    navigator.geolocation.getCurrentPosition(showPosition);

  } else {

    x.innerHTML = "Geolocation is not supported by this browser.";

  }

}

function showPosition(position) {

  x.innerHTML = "Latitude: " + position.coords.latitude +

  "<br>Longitude: " + position.coords.longitude;

}

</script>
```

**Handling Errors and Rejections: -**

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.

```
function showError(error) {
  switch(error.code) {
    case error.PERMISSION_DENIED:
      x.innerHTML = "User denied the request for Geolocation."
      break;
    case error.POSITION_UNAVAILABLE:
      x.innerHTML = "Location information is unavailable."
      break;
    case error.TIMEOUT:
      x.innerHTML = "The request to get user location timed out."
      break;
    case error.UNKNOWN_ERROR:
      x.innerHTML = "An unknown error occurred."
      break;
  }
}
```

**Displaying the Result in a Map: -**

To display the result in a map, you need access to a map service, like Google Maps.

```
function showPosition(position) {

  let latlon = position.coords.latitude + "," + position.coords.longitude;

  let img_url = "https://maps.googleapis.com/maps/api/staticmap?center=

  "+latlon+"&zoom=14&size=400x300&sensor=false&key=YOUR_KEY";

  document.getElementById("mapholder").innerHTML = "<img src='"+img_url+"'>";

}
```

**The getCurrentPosition() Method: -**

| **Property** | **Returns** |
|---|---|
| coords.latitude | The latitude as a decimal number (always returned) |
| coords.longitude | The longitude as a decimal number (always returned) |
| coords.accuracy | The accuracy of position (always returned) |
| coords.altitude | The altitude in meters above the mean sea level (returned if available) |
| coords.altitudeAccuracy | The altitude accuracy of position (returned if available) |
| coords.heading | The heading as degrees clockwise from North (returned if available) |
| coords.speed | The speed in meters per second (returned if available) |
| timestamp | The date/time of the response (returned if available) |

**Geolocation Object: -**

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.

```
<script>
const x = document.getElementById("demo");
function getLocation() {
  if (navigator.geolocation) {
    navigator.geolocation.watchPosition(showPosition);
  } else {
    x.innerHTML = "Geolocation is not supported by this browser.";
  }
}
function showPosition(position) {
  x.innerHTML = "Latitude: " + position.coords.latitude +
  "<br>Longitude: " + position.coords.longitude;
}
</script>
```