ABES Engineering College

FSD Central Training Team

Advance JavaScript Notes

Table of Content

1. What is JavaScript?

JavaScript was initially created to “make web pages alive”.

The programs in this language are called *scripts*. They can be written right in a web page’s HTML and run automatically as the page loads.

Scripts are provided and executed as plain text. They don’t need special preparation or compilation to run.

In this aspect, JavaScript is very different from another language called Java.

The browser has an embedded engine sometimes called a “JavaScript virtual machine”.

Different engines have different “codenames”. For example:

- **V8 – in Chrome, Opera and Edge.**
- **SpiderMonkey – in Firefox.**

2. Variable in JS

A variable is a "named storage" for data. We can use variables to store goodies, visitors, and other data.

To create a variable in JavaScript, use the let keyword.

```
let message;

message = 'Hello'; // store the string 'Hello' in the variable named message
```

We can also declare multiple variables in one line:

```
let user = 'John', age = 25, message = 'Hello';
```

We can also change it as many times as we want:

```
let message;

message = 'Hello!';

message = 'World!'; // value changed
```

Declaring twice triggers an error

A variable should be declared only once.

A repeated declaration of the same variable is an error:

```
let message = "This";

// repeated 'let' leads to an error
let message = "That"; // SyntaxError: 'message' has already been declared
```

3.Variable naming

There are two limitations on variable names in JavaScript:

- **The name must contain only letters, digits, or the symbols `$` and _.**
- **The first character must not be a digit.**

These names are valid:

```
let $ = 1; // declared a variable with the name "$"
let _ = 2; // and now a variable with the name "_"

alert($ + _); // 3
```

4.Reserved names

There is a list of reserved words, which cannot be used as variable names because they are used by the language itself.

For example: let, class, return, and function are reserved.

The code below gives a syntax error:

```
let let = 5; // can't name a variable "let", error!
let return = 5; // also can't name it "return", error!
```

5.An assignment without use strict

Normally, we need to define a variable before using it. But in the old times, it was technically possible to create a variable by a mere assignment of the value without using let. This still works now if we don’t put use strict in our scripts to maintain compatibility with old scripts.

```
// note: no "use strict" in this example

num = 5; // the variable "num" is created if it didn't exist

alert(num); // 5
```

```
"use strict";

num = 5; // error: num is not defined
```

6. Constants

To declare a constant (unchanging) variable, use const instead of let:

```
const myBirthday = '18.04.1982';
```

Variables declared using const are called “constants”. They cannot be reassigned. An attempt to do so would cause an error:

```
const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // error, can't reassign the constant!
```

Summary

We can declare variables to store data by using the var, let, or const keywords.

let – is a modern variable declaration.
var – is an old-school variable declaration. Normally we don’t use it at all, but we’ll cover subtle differences from let in the chapter The old "var", just in case you need them.
const – is like let, but the value of the variable can’t be changed.

7.Type Conversions

Most of the time, operators and functions automatically convert the values given to them to the right type.

For example, alert automatically converts any value to a string to show it.

Mathematical operations convert values to numbers.

There are also cases when we need to explicitly convert a value to the expected type.

```
let value = true;
alert(typeof value); // boolean

value = String(value); // now value is a string "true"
alert(typeof value); // string
```

```
alert( "6" / "2" ); // 3, strings are converted to numbers
let str = "123";
alert(typeof str); // string

let num = Number(str); // becomes a number 123

alert(typeof num); // number
```

Boolean Conversion

Boolean conversion is the simplest one.

It happens in logical operations (later we'll meet condition tests and other similar things) but can also be performed explicitly with a call to Boolean(value).

The conversion rule:

- **Values that are intuitively "empty", like 0, an empty string, null, undefined, and NaN, become false.**
- **Other values become true.**

For instance:

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false
alert( Boolean("hello") ); // true
alert( Boolean("") ); // false
```

8. Interaction: alert, prompt, confirm

- **alert**

shows a message.

- **prompt**

shows a message asking the user to input text. It returns the text or, if Cancel button or Esc is clicked, null.

- **confirm**

shows a message and waits for the user to press “OK” or “Cancel”. It returns true for OK and false for Cancel/Esc.

A strict equality operator === checks the equality without type conversion.

In other words, if a and b are of different types, then a === b immediately returns false without an attempt to convert them.

9.Arrow functions

There's another very simple and concise syntax for creating functions, that's often better than Function Expressions.
It's called "arrow functions", because it looks like this:

```
let func = (arg1, arg2, ..., argN) => expression;
```

This creates a function func that accepts arguments arg1..argN, then evaluates the expression on the right side with their use and returns its result.
In other words, it's the shorter version of:

```
let func = function(arg1, arg2, ..., argN) {
  return expression;
};
```

```
let sayHi = () => alert("Hello!");

sayHi();
```

Multiline arrow functions

The arrow functions that we've seen so far were very simple. They took arguments from the left of `=>`, evaluated and returned the right-side expression with them.

Sometimes we need a more complex function, with multiple expressions and statements. In that case, we can enclose them in curly braces. The major difference is that curly braces require a `return` within them to return a value (just like a regular function does).

Like this:

```
let sum = (a, b) => {  // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, then we need an explicit "return"
};

alert( sum(1, 2) );
```

10.DOM in JS

Common methods to access DOM elements include:

document.getElementById(id):
Selects an element by its id.

javascript
-->
const header = document.getElementById('header');

document.getElementsByClassName(class):
Selects all elements with a specific class name (returns an HTMLCollection).

javascript
-->
const paragraphs = document.getElementsByClassName('text');

document.getElementsByTagName(tag):
Selects all elements with a specific tag (returns an HTMLCollection).

javascript
-->
const divs = document.getElementsByTagName('div');

document.querySelector(selector):
Selects the first element that matches a CSS selector.

javascript
-->
const firstParagraph = document.querySelector('p');

document.querySelectorAll(selector):

Selects all elements that match a CSS selector (returns a NodeList).

```
javascript
-->
const allParagraphs = document.querySelectorAll('p');
```

11.Manipulating the DOM

Once elements are accessed, you can manipulate them in various ways, such as changing their content, attributes, or styles.

1. Changing Text Content

```
javascript
-->
const header = document.querySelector('h1');
header.textContent = 'New Heading';
```

2. Changing HTML Content

```
javascript
-->
const paragraph = document.querySelector('p');
paragraph.innerHTML = '<strong>This is bold text.</strong>';
```

3. Changing Attributes

```
javascript
-->
const link = document.querySelector('a');
link.setAttribute('href', 'https://www.example.com');
```

4. Changing Styles

```
javascript
-->
const div = document.querySelector('div');
div.style.backgroundColor = 'lightblue';
div.style.fontSize = '20px';
```

12.Creating and Appending Elements

We can dynamically create new elements and add them to the DOM.

1. Creating a New Element

```
const newDiv = document.createElement('div');
newDiv.textContent = 'This is a new div';
```

2. Appending an Element to the DOM

```
const body = document.querySelector('body');
body.appendChild(newDiv);
```

13.Event Handling in the DOM

JavaScript can be used to respond to user interactions with DOM elements by attaching event listeners.

Example: Adding a Click Event Listener

```
const button = document.querySelector('button');
button.addEventListener('click', function() {
  alert('Button clicked!');
});
```

Example: DOM Manipulation

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>DOM Example</title>
</head>
<body>
 <h1 id="title">Original Title</h1>
 <button id="changeTitleBtn">Change Title</button>

 <script>
  const titleElement = document.getElementById('title');
  const button = document.getElementById('changeTitleBtn');

  button.addEventListener('click', () => {
   titleElement.textContent = 'Title Changed!';
  });
 </script>
</body>
</html>
```

In this example, clicking the button changes the text content of the <h1> element.

```
<div id="rectdiv"></div>
  <button onclick="alert('welocme')">Click</button>
  <button id="btn">Button 1</button>
  <button id="btn1">Button 2</button>
  <button id="btn3">Button 3</button>

  <script>
    document.getElementById('btn').onclick = () => {
      alert('Welocme to JS');
    };
```

```
    document.getElementById('btn').onmouseover = () => {
      alert('Welocme to JS');
    };

    let btn1 = document.getElementById('btn1');
    btn1.addEventListener('click', () => {
      alert('Welcome to JS with functions');
    })
    const handleclick = () => {
      let rectdiv = document.getElementById('rectdiv');
      rectdiv.style.translate = '30rem 0';
      alert("hello");
    }

    let btn3 = document.getElementById('btn3');
    btn3.addEventListener('click', handleclick);
  </script>
```

```
<input type="text" id="myinput" placeholder="type here..">
  <div class="output">
    <p class="keypressed">KeyPressed</p>
  </div>

  <script>
    const handlekeypress = (event) => {
      document.querySelector('.keypressed').textContent = `Key Pressd
${event.target.value}`;

    }
    let myinput = document.getElementById('myinput');
    myinput.addEventListener('keydown', handlekeypress);
  </script>
```

14.MAP, FILTER, REDUCE

In JavaScript, the map(), filter(), and reduce() methods are array functions that allow you to manipulate and process data effectively. Each of these methods is used for a different purpose, but together they are extremely powerful for handling data collections.

1. map() Method

The map() method creates a new array by calling a function on every element in the calling array. It does not modify the original array.

Syntax:

```
let newArray = array.map(callback(currentValue, index, array));
```

Example:

javascript code

```
const numbers = [1, 2, 3, 4];
const squaredNumbers = numbers.map(num => num * 2);

console.log(squaredNumbers); // [2, 4, 6, 8]
```

2. filter() Method

The filter() method creates a new array with all the elements that pass the test implemented by the provided function. Like map(), it does not modify the original array.

Syntax:

```
let newArray = array.filter(callback(currentValue, index, array));
```

Example:

javascript code

```
const numbers = [1, 2, 3, 4, 5, 6];
const evenNumbers = numbers.filter(num => num % 2 === 0);

console.log(evenNumbers); // [2, 4, 6]
```

3. reduce() Method

The reduce() method applies a function to an accumulator and each element in the array (from left to right) to reduce the array to a single value.

Syntax:

javascript code

```
let result = array.reduce(callback(accumulator, currentValue, index, array), initialValue);
```

accumulator: The accumulated value from previous iterations.
currentValue: The current element being processed.

Example:

javascript code

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);

console.log(sum); // 10
```

Problem: Given an array of student objects, find the total score of students who scored more than 60, after multiplying their score by 2.

Javascript code

```
const students = [
   { name: 'Alice', score: 50 },
   { name: 'Bob', score: 65 },
   { name: 'Charlie', score: 80 },
   { name: 'David', score: 45 },
];
```

// Using map(), filter(), and reduce() together

```
const totalScore = students
   .filter(student => student.score > 60)  // Step 1: Filter students with scores > 60
   .map(student => student.score * 2)      // Step 2: Multiply their scores by 2
   .reduce((accumulator, score) => accumulator + score, 0); // Step 3: Sum the scores

console.log(totalScore); // Output: 290
```

Summary:

map(): Transforms each element of an array based on the provided function.

filter(): Returns a new array with elements that pass a condition.

reduce(): Reduces the array to a single value by applying a function.

```
let arr = [1, 2, 32, 45, 4, 33, 3, 3];
let newar = arr.map((x) => Math.sqrt(x));
console.log(newar);
console.log(typeof arr)
```

```
let kv = [
  {
    key:'one',val:1,
  },
  {
    key:'two',val:2,
  },
];
newkv=kv.map(({key,val})=>({[key]:val*3}))
console.log(newkv)
```

```
const words=['I','am',"working",'as','corporate','trainer'];
const len=words.filter(word=>word.length>7);
```

```
console.log(len);
```

```
const words=['I','am',"working",'as','corporate','trainer'];
console.log(words.copyWithin(3,2,4))
```

['I', 'am', 'working', 'working', 'as', 'trainer']

```
const words=['I','am',"working",'as','corporate','trainer'];
let newwords=words.entries();
for(let i=0;i<words.length;i++)
{
  console.log(newwords.next().value)
}
```

```
[ 0, 'I' ]
[ 1, 'am' ]
[ 2, 'working' ]
[ 3, 'as' ]
[ 4, 'corporate' ]
[ 5, 'trainer' ]
```

```
const arr=[1,2,2,3,[3,4,3,4,4,56]];
const newarr=arr.flat();
console.log(newarr)
```

```
[
  1, 2, 2, 3,  3,
  4, 3, 4, 4, 56
]
```

```
const arr=[1,2,2,3,9];
arr2=arr.forEach(x=>(console.log(x*2)));
```

```
PS C:\Users\User\Desktop\JS_Notes> node .\p1.js
2
4
4
6
18
```

```
let sum = (a, b) => {
  return a + b;
};

console.log(sum(12, 23));
```

15.Callback Functions

In JavaScript, a callback **is a function that is passed as an argument to another function and is executed after that function completes its operation.**
Callbacks are used to handle asynchronous operations, such as API calls, file reading, or event handling, ensuring that specific tasks are executed after others have finished.

Synchronous Callback:

```
function greet(name) {
   console.log('Hello ' + name);
}
function processUserInput(callback) {
   const name = 'John';
   callback(name);
}
processUserInput(greet);  // Output: Hello John
```

The greet function is passed as a callback to processUserInput, and it's executed after processUserInput finishes.

Asynchronous Callback (using setTimeout):

```
function sayHello() {
   console.log('Hello!');
}

console.log('Start');
setTimeout(sayHello, 2000);  // Will execute after 2 seconds
console.log('End');
```

Output:

```
Start
End
Hello!
```

sayHello is executed after a 2-second delay due to the setTimeout function. The callback (sayHello) runs asynchronously after the main program completes its execution.

NOTE: Callbacks are essential for handling asynchronous operations like API requests, timers, or event listeners.

16. Callback hell

refers to a situation where several nested callbacks are used in a way that makes the code difficult to read and maintain. It often occurs when asynchronous operations are chained together, leading to deeply nested code that becomes difficult to follow.

```
console.log("start");
setTimeout(() => {
  console.log("first task complted");
  setTimeout(() => {
    console.log("second task completed");
    setTimeout(() => {
      console.log("third task completed");
    }, 3000);
  }, 2000);
}, 1000);
console.log("End");
```

In this example, each setTimeout callback is nested inside the previous one, creating a structure that is hard to read and maintain. This is what is referred to as "callback hell."

Problems with Callback Hell:

- **Difficult to read and understand: The more nested the callbacks become, the harder it is to follow the flow of the program.**
- **Error handling becomes complex: If an error occurs, managing it in deeply nested callbacks can become messy.**
- **Maintenance issues: If changes are required, updating the code can be cumbersome and prone to introducing new errors.**

Solutions to Callback Hell:

- **Promises: Promises allow handling asynchronous operations in a more linear, readable way.**
- **Async/Await: This is a modern way to write asynchronous code in a synchronous-looking manner, further simplifying the code.**

17.Promises in JavaScript

A promise in JavaScript is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value. Promises allow you to handle asynchronous code in a more readable and structured way.

Promise States:

- **Pending: Initial state, neither fulfilled nor rejected.**
- **Fulfilled: Operation completed successfully.**
- **Rejected: Operation failed.**

Basic Example of a JavaScript Promise:

Javascript code

```
// Creating a new promise
const myPromise = new Promise((resolve, reject) => {
  let success = true; // Change this to false to simulate failure

  if (success) {
    resolve("The operation was successful!"); // Fulfilled
  } else {
    reject("The operation failed."); // Rejected
  }
});

// Handling the promise with .then() and .catch()
myPromise
  .then((message) => {
    console.log("Success: " + message);
  })
  .catch((error) => {
    console.log("Error: " + error);
```

```
});
```

A promise is created using new Promise((resolve, reject) => {...}).
Inside the promise, you either call resolve to indicate success or reject to indicate failure.
.then() is used to handle the success scenario (fulfilled).
.catch() is used to handle the failure scenario (rejected).

Example with Asynchronous Code:

Here's an example using setTimeout to simulate an asynchronous operation:

javascript code

```
// Simulating an asynchronous operation with setTimeout
const fetchData = new Promise((resolve, reject) => {
  console.log("Fetching data...");

  setTimeout(() => {
    let dataReceived = true; // Change to false to simulate an error

    if (dataReceived) {
      resolve("Data successfully fetched!");
    } else {
      reject("Failed to fetch data.");
    }
  }, 2000); // Simulate a 2-second delay
});

fetchData
  .then((message) => {
    console.log(message);
  })
  .catch((error) => {
    console.log(error);
```

});

Key Points:

Promise chaining: You can chain multiple .then() calls to handle sequential asynchronous operations.

Error handling: Use .catch() to handle errors. You can also use .finally() to execute code after the promise is settled (fulfilled or rejected), regardless of the outcome.

Example of Promise Chaining:

javascript code

```
function task(message, delay) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(message);
      resolve();
    }, delay);
  });
}

task("First Task completd", 1000)
  .then(() => task("Second task is complted", 2000))
  .then(() => task("third task is completd", 3000));
```

18. Fetching Data using Promise

```
function fetchdata() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = { id: 1, roll: 23, name: "rahul" };
      resolve(data);
    }, 3000);
    //reject("Error fetching data");
  });
}
fetchdata()
  .then((data) => {
    console.log("Data received:", data); // Output: { id: 1, name: 'John Doe' }
  })
  .catch((error) => {
    console.error("Error:", error); // Handle any potential errors
  });
```

19. async/await is a modern JavaScript feature introduced in ES2017 (ES8) that allows for writing asynchronous code in a more synchronous and readable manner.

It is built on top of Promises and provides a cleaner, more elegant way to handle asynchronous operations compared to the traditional use of callbacks or .then() chaining with Promises.

Key Concepts:

async:

A function declared with the async keyword automatically returns a Promise.

Even if the function doesn't explicitly return a Promise, JavaScript wraps the return value in a Promise.

await:

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

It makes JavaScript wait for a Promise to resolve or reject before continuing execution of the next line of code.

While waiting, the code execution inside the function pauses, but the program continues executing the rest of the code outside of the async function (non-blocking behavior).

```
async function orderFood(name, time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(name + "Prepared");
    }, time);
  });
}
async function restaurant() {
  console.log("Order Placed");
  const pizza = await orderFood("pizza", 1000);
  console.log(pizza);
  const burger = await orderFood("burger", 2000);
  console.log(burger);
  const pasta = await orderFood("pasta", 3000);
  console.log(pasta);
  console.log("All Food Prepared");
}

restaurant();
```

reject if burger is not available

```
async function orderFood(name, time) {
  return new Promise((resolve, reject) => {
    if (name == "burger") reject("Burger Not Available");
    setTimeout(() => {
      resolve(name + "Prepared");
    }, time);
  });
}

async function restaurant() {
  console.log("Order Placed");
  try {
    const pizza = await orderFood("pizza", 1000);
    console.log(pizza);
  } catch (err) {
    console.log(err);
  }

  try {
    const burger = await orderFood("burger", 2000);
    console.log(burger);
  } catch (error) {
    console.log(error);
  }

  try {
    const pasta = await orderFood("pasta", 3000);
    console.log(pasta);
  } catch (error) {}

  console.log("All Food Prepared");
}

restaurant();
```

20. Fetch Top 10 Github users using async/await.

Fetch login, node_id and url of top 10 github users on console.

```
async function fetchdata() {
  try {
    let response = await fetch("https://api.github.com/users?per_page=10");
    let data = await response.json();

    // Extract and log login, node_id, and url from each user object
    data.forEach(user => {
      console.log(`Login: ${user.login}, Node ID: ${user.node_id}, URL: ${user.url}`);
    });

  } catch (error) {
    console.error("Error fetching data:", error);
  }
}

// Call the async function
fetchdata();
```

Fetch and display data of top 10 GitHub users on web page index.html.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="script.js"></script>
</head>

<body>
  <button onclick="fetchdata()">Click Here</button>
  <h1>Git Hub Top 10 users</h1>
  <div id="top"></div>
</body>

</html>
```

```
async function fetchdata() {
  let response = await fetch("https://api.github.com/users?per_page=10");
  let data = await response.json();
  data.forEach((element) => {
    let h1 = document.createElement("h1");
    h1.textContent = element.login;
    let a = document.createElement("a");
    a.textContent = element.login;
    a.setAttribute("href", element.html_url);
    let top = document.getElementById("top");
    top.appendChild(h1);
    top.appendChild(a);
  });
}
```