

# JS\_Beginning\_to\_Mastery\_Part2\_1

April 21, 2023

## 1 How JavaScript works?

JavaScript is an interpreted language, which means that the code is executed directly by an interpreter, rather than being compiled into machine code beforehand like in traditional compiled languages such as C or Java.

When you run JavaScript code in a web browser or on the server side with Node.js, the code is interpreted by the JavaScript engine in the runtime environment. The interpreter reads the code line by line, converts it to machine code, and executes it.

However, there are some tools and techniques that can be used to compile JavaScript code for various purposes, such as optimizing performance or transpiling modern JavaScript syntax to older syntax that is more widely supported. These tools include Babel, which is a popular tool for transpiling modern JavaScript code to older syntax, and various tools for minifying and optimizing JavaScript code.

In summary, JavaScript is primarily an interpreted language, but there are also tools and techniques available for compiling and optimizing JavaScript code.

Compilation is the process of translating human-readable source code into machine-executable code that a computer can understand.

In compiled languages, the source code is translated into machine code beforehand, which is then executed by the computer. This machine code can be directly executed by the computer's CPU, without requiring any additional translation or interpretation.

Compiled languages typically offer better performance and security than interpreted languages, because the code is optimized and checked for errors before it is executed. However, they can also be more difficult to develop in, because the development process requires the additional step of compiling the code before it can be tested or executed.

Examples of compiled languages include C, C++, Java, and Rust.

In contrast, interpreted languages, such as JavaScript and Python, are executed directly by an interpreter, which reads and executes the code line by line, without requiring any prior compilation. Interpreted languages typically offer faster development times and greater flexibility, but may sacrifice some performance and security.

JavaScript is a high-level, dynamically-typed, interpreted programming language that is primarily used to create interactive web pages and web applications. It is executed by a JavaScript engine, which is a software component that reads and executes JavaScript code.

When a web page containing JavaScript code is loaded in a web browser, the JavaScript engine parses the code and executes it line by line. It can also interact with other parts of the web page, such as the Document Object Model (DOM) and the browser's window object, in order to create interactive functionality.

JavaScript code can also be executed on the server side using the Node.js runtime environment, which allows JavaScript to be used for creating back-end web applications and services.

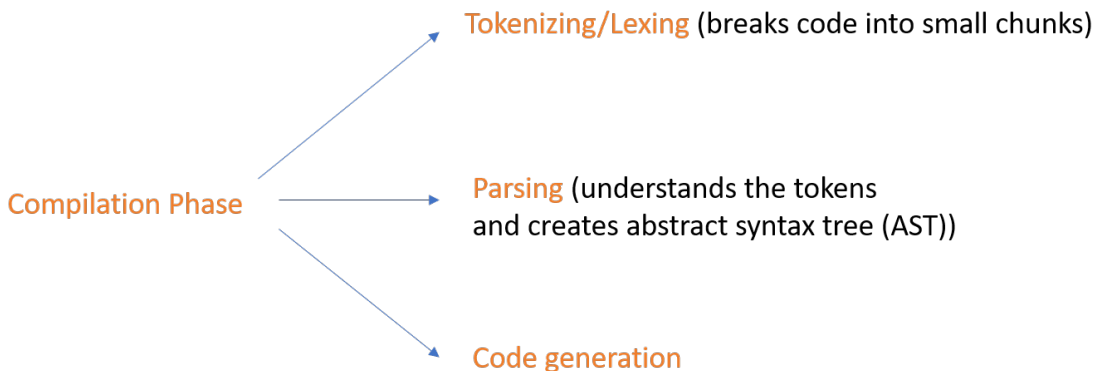
Some key features of JavaScript include its ability to manipulate web page content and styles in real time, handle user input and events, and interact with server-side resources via asynchronous requests such as XMLHttpRequests or the newer Fetch API. It also supports a wide range of programming paradigms, including object-oriented, functional, and procedural programming.

In recent years, JavaScript has also expanded beyond its original use case of client-side web development, and is now widely used for server-side development, desktop and mobile application development, and even for creating complex Internet of Things (IoT) devices.

JavaScript does the following steps sequentially:

1. Compile (first compile the code)/parse
2. Code execute

### 1.0.1 Compilation phase



**Note:** JS is also called lexical scoped language.

In original ES documentation, it is written that:

1. Early error checking needs to be done before code execution
2. Determining appropriate scope for variables

But to do these 2, we have to parse the code. So different browsers have different techniques for parsing the JS code or for compiling the code.

So, before the first code is executed, the code is first compiled.

Suppose our code is like below:

```
console.log(this);  
console.log(window);  
console.log(firstName);  
var firstName = "John";
```

Now, even before executing the first line, this will throw error, as the last line has syntax error (extra .). Due to the parsing of all the code, this error is caught.

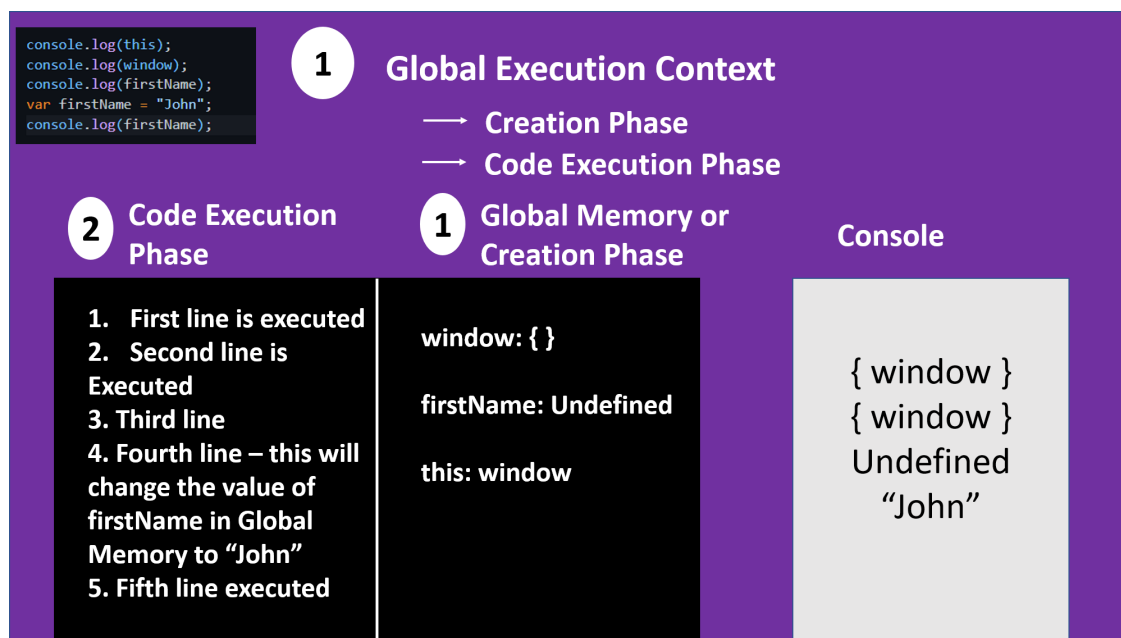


### 1.0.2 Code execution

In JS, code executes inside **execution context**.

Before executing first line, first Global Execution Context is created. It has two parts:

- Creation phase (sets some variables to global memory)
- Code execution phase



If we had used `let` to define variable `firstName`, then it would have the value `uninitialized` in Global Memory or in the Creation Phase of the Global Execution Context. Thus, calling this or printing this before initialization will give `Uncaught ReferenceError`.

JS is synchronous programming language. When first line is executed, the execution of second line will not start unless the first line gets finished.

Browser gives us the feature of asynchronous programming. It is single threaded.

### 1.0.3 Hoisting

Hoisting in JavaScript is a mechanism that allows variable and function declarations to be moved to the top of their respective scope at the compile time, regardless of where they were declared within the scope. This means that you can use a variable or a function before it has been declared, without getting a reference error.

In more technical terms, hoisting is the behavior of the JavaScript interpreter where it moves all variable and function declarations to the top of the current scope before executing any code. This is done to ensure that the code runs correctly, as JavaScript is an interpreted language that executes code line by line.

```
[1]: console.log(a);  
     var a = 10;
```

undefined

At first glance, it might seem like this code would throw an error because `a` is being used before it has been declared. However, due to hoisting, the variable declaration is moved to the top of the scope, so the code is actually interpreted as:

```
var a;  
console.log(a);  
a = 10;
```

This means that `a` is defined as `undefined` when it is logged to the console, rather than throwing an error. It's important to note that only the declaration is hoisted, not the initialization, so if you try to use a variable that has been declared but not initialized, it will still be `undefined`.

It's also important to note that hoisting only affects variable and function declarations, not variable assignments or function expressions. So if you try to use a variable that has been declared with `let` or `const` before it has been initialized, you will get a `ReferenceError`. Similarly, function expressions are not hoisted, so if you try to use a function expression before it has been defined, you will also get a `ReferenceError`. Same error will also occur if we call variable declared with `const` before initialization.

#### For function declaration

```
console.log(myFunction);  
  
// function declaration  
function myFunction(){  
    console.log("this is my function");  
}  
  
console.log(myFunction);
```

The output in the console will be

```

f myFunction(){
  console.log("this is my function");
}
f myFunction(){
  console.log("this is my function");
}
>

```

[file72.js:1](#)  
[file72.js:8](#)

### For function expression

```
console.log(myFunction);
```

```

// function expression
var myFunction = function(){
  console.log("this is my function");
}

```

```
console.log(myFunction);
```

The output in the console will be

```

undefined
f (){
  console.log("this is my function");
}
> |

```

[file71.js:1](#)  
[file71.js:8](#)

```

[2]: let foo = "foo";
     console.log(foo);

function getFullName(firstName, lastName) {
  // arguments is array-like object which has index and length
  console.log(arguments); // we can also use arguments[0] or arguments[1]
  // we can also use arguments.length
  let myVar = "var inside func";
  console.log(myVar);
  const fullName = firstName + " " + lastName;
  return fullName;
}

const personName = getFullName("John", "Doe");
console.log(personName);

```

```

foo
[Arguments] { '0': 'John', '1': 'Doe' }
var inside func
John Doe

```

### 1.0.4 Lexical environment, scope chain

```
// lexical environment, scope chain
```

```
const lastName = "Doe";

const printName = function(){
  const firstName = "John";

  console.log(firstName);
  console.log(lastName);
}
printName();
```

The output in the console will be

John	<a href="#">file81.js:9</a>
Doe	<a href="#">file81.js:10</a>

[3]: *// lexical environment, scope chain*

```
const lastName = "Doe";

const printName = function(){
  const firstName = "John";

  function myFunction(){
    console.log(firstName);
    console.log(lastName);
  }
  myFunction();
}
printName();
```

John  
Doe

The output in the console will again be the same

John	<a href="#">file81.js:9</a>
Doe	<a href="#">file81.js:10</a>

### 1.0.5 Closures

We need to know that functions can return functions.

[4]:

```
function outerFunction(){
  function innerFunction(){
    console.log("innerFunction");
  }
  return innerFunction;
}
```

```
const ans = outerFunction();
ans();
```

innerFunction

```
[7]: function printFullName3(firstName, lastName){
      function printName(){
        console.log(firstName, lastName);
      }
      return printName;
    }

    const ans3 = printFullName3("John", "Doe");
    ans3();
```

John Doe

When a function inside a function is returned, here, when `printName()` is returned from inside `printFullName3()`, it gets returned with variables in the local memory, or the variables `firstName` and `lastName`. These variables `firstName` and `lastName` are in the closures.

Inner functions can access outer function elements.

```
[9]: function hello4(x){
      const a = "varA";
      const b = "varB";
      return function(){
        console.log(a, b, x);
      }
    }

    const ans4 = hello4("arg");
    ans4();
```

varA varB arg

```
[12]: function myFunction5(power){
      return function(number){
        return number ** power;
      }
    }

    const cube5 = myFunction5(3);
    const ans5 = cube5(2);
    console.log(ans5);
```

8

```
[13]: myFunction5(2)(8)
```

[13]: 64

Another application for closure – Working differently if called more than once or not even working

```
[17]: function func7(){
      let counter = 0;
      return function(){
        if(counter < 1){
          console.log("Hi! Thanks for calling me the first time!");
          counter++;
        }
        else{
          console.log("I have already been called once!");
        }
      }
    }

    const myFunc7 = func7();
```

```
[18]: myFunc7();
```

Hi! Thanks for calling me the first time!

```
[19]: myFunc7();
```

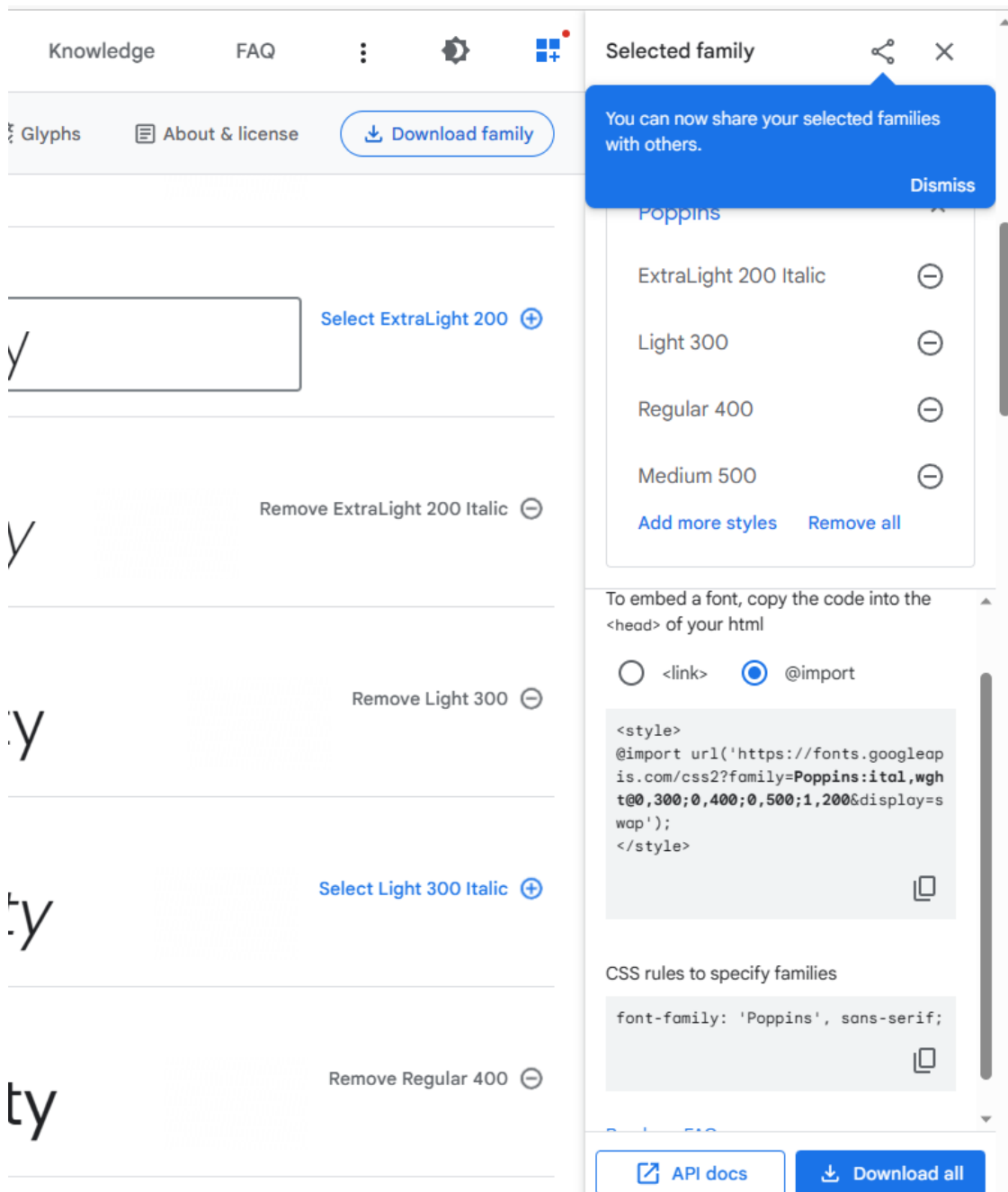
I have already been called once!

### 1.0.6 Working with DOM

For selecting different fonts, we can use this website: <https://fonts.google.com/specimen/Poppins>

After selecting different fonts, we can copy the `@import url(...` section of the code shown in the bottom right corner.





In the CSS file, after copying the `@import url(...` part, copy also the part written under the section **CSS rules to specify families**. Put this inside `body{}` tag as below:

```
body{
  font-family: 'Poppins', sans-serif;
}
```

For selecting nice background, we can use [pexels.com](https://pexels.com) and search for wallpaper.

**VSCode shortcut** To write the below html code:

```
<section class="section-signup"></section>
```

In VSCode, write `section.section-signup` and then press TAB to autocomplete this line.

### 1.0.7 Linking JS to HTML file

**1st way** If we link our js file inside `<head>` tag in HTML, it will throw error if the JS is using the HTML elements below as the HTML elements used in this file below are not read or loaded yet.

Don't do the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="./style.css">
  <!--
    We can link our js file here using below:
    <script src="./102.js"></script>
    But this will throw error if the JS is using the HTML elements below
    as the HTML elements used in this file below are not read or loaded yet
  -->
  <script src="./102.js"></script>
</head>
<title>My Website</title>
```

**2nd way** A better approach will be to use it just before the end of `<body>` tag.

Do the following:

```
</form>
</section>
<script src="./102.js"></script>
</body>
```

But even this latter option also has some issues. In this case, first HTML part will be parsed, then JS will load, and then JS will be executed. These will take some time.

**3rd way** Third approach to load JS is to put script inside `<head>` tag and before `<title>` tag as before but adding `async` like this: `<script src="./102.js" async></script>`. Have a look at example code below:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="./style.css">
  <script src="./102.js" async></script>
  <!--
  -->
</head>
<title>My Website</title>
```

*We can link our js file here using below:*

```
<script src="./102.js"></script>
```

*But this will throw error if the JS is using the HTML elements below  
as the HTML elements used in this file below are not read or loaded yet*

```
-->
```

```
<!--
```

```
<script src="./102.js"></script>
```

```
-->
```

```
<script src="./102.js" async></script>
```

```
<title>My Website</title>
```

```
</head>
```

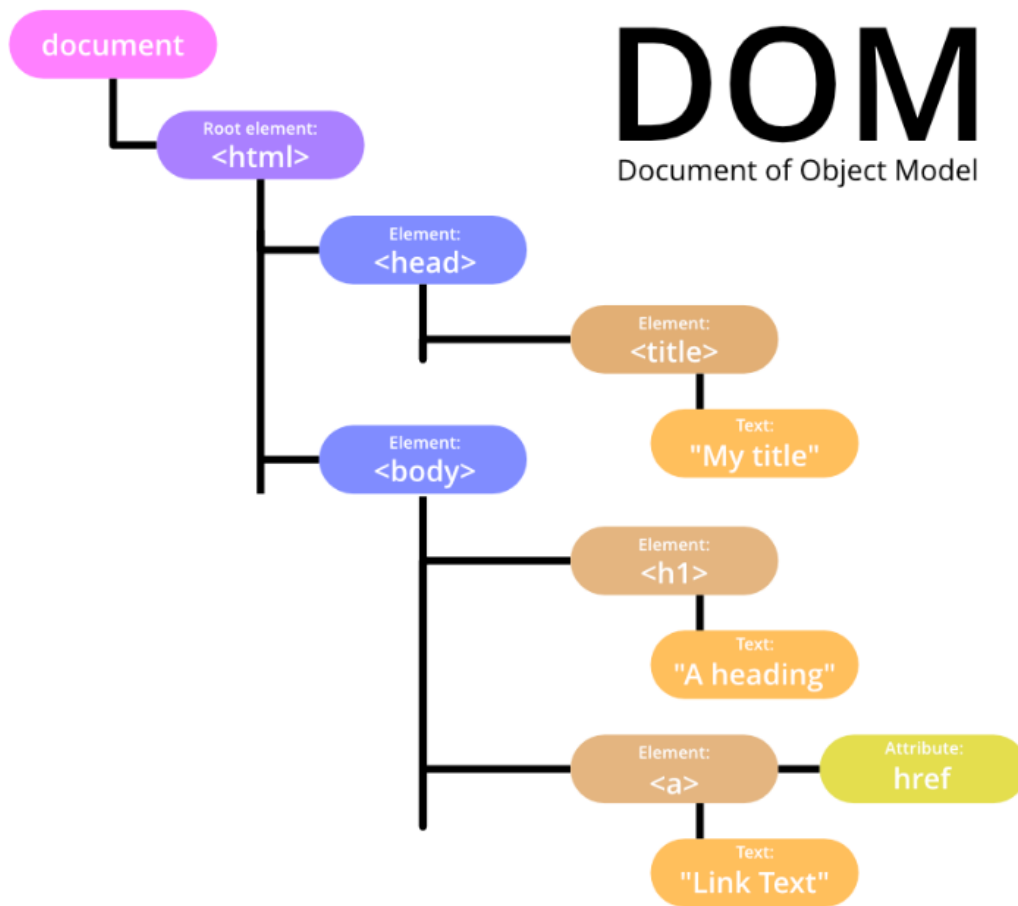
This will allow to continue parsing HTML even after reaching the `<script>` tag while also continuing the load of JS file. Parsing HTML and loading JS will go together in this case. But the problem is as soon as the JS file is loaded, parsing of HTML file will stop and the execution of JS file will start. The probability of error coming is very high as the full HTML file is not parsed yet.

**4th way – the best approach** Put script inside `<head>` tag and before `<title>` tag as before but adding `defer` like this: `<script src="./102.js" defer></script>`

This will allow to continue parsing HTML even after reaching the `<script>` tag while also continuing the load of JS file. Parsing HTML and loading JS will go together in this case. Compared to the previous approach, in this case as soon as the JS file is loaded, parsing of HTML file will not stop and the execution of JS file will start after the parsing of HTML file is completed.

This will increase the performance of the website. No chance of error in this case.

### 1.0.8 Document Object Model (DOM)



Source:

The above figure is taken from Geeks for Geeks website

Will create object of key-value pairs. This object is called `document`. Then browser will add `document` inside `window` object residing inside JS.

Now, we have `index.html` as following:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="./style.css">
<!--
```

*We can link our js file here using below:*

```
<script src="./102.js"></script>
```

*But this will throw error if the JS is using the HTML elements below as the HTML elements used in this file below are not read or loaded yet*

```

-->
<!--
    1st approach:
    <script src="./102.js"></script>
-->
<!--
    3rd approach:
    <script src="./102.js" async></script>
-->

<!--
    4th approach: best approach using defer
    <script src="./102.js" defer></script>
-->
<script src="./102.js" defer></script>

<title>My Website</title>
</head>
<body>
    <header class="header">
        <nav class="nav container">
            <h1 class="log">Website</h1>
            <ul class="nav-items">
                <li><a href="">Home</a></li>
                <li><a href="">To Do</a></li>
                <li><a href="">Sign In</a></li>
            </ul>
        </nav>
        <div class="headline">
            <h2>Manage your tasks</h2>
            <button class="btn btn-headline">Learn More</button>
        </div>
    </header>
    <section class="section-todo container">
        <h2>What do you plan to do today?</h2>
        <form class="form-todo">
            <input type="text" name="" id="" placeholder="Add Todo">
            <input type="submit" value="Add Todo" class="btn">
        </form>
    </section>

    <section class="section-signup container">
        <h2>Sign Up</h2>
        <form class="signup-form">
            <div class="form-group">
                <label for="username">Username</label>
                <input type="text" name="username" id="username">
            </div>

```

```

    <div class="form-group">
        <label for="password">password</label>
        <input type="password" name="password" id="password">
    </div>
    <div class="form-group">
        <label for="confirmPassword">Confirm Password</label>
        <input type="password" name="confirmPassword" id="confirmPassword">
    </div>
    <div class="form-group">
        <label for="email">Email</label>
        <input type="email" name="email" id="email">
    </div>
    <div class="form-group">
        <label for="about">About Yourself</label>
        <textarea name="about" id="about" cols="30" rows="10"></textarea>
    </div>
    <button type="submit" class="btn signup-btn">Submit</button>

</form>
</section>
<!--
    2nd approach: Putting before the end of <body> tag
    <script src="./102.js"></script>
-->
</body>
</html>

```

We also have style.css as following:

```

@import url('https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,300;0,400;0,500;600;');

*{
    box-sizing: border-box;
    padding: 0;
    margin: 0;
}

body{
    font-family: 'Poppins', sans-serif;
}

a{
    text-decoration: none;
    color: white;
}

.header{
    position: relative;
    min-height: 60vh;

```

```

    background: url('../background.jpg');
    background-position: center;
    background-repeat: no-repeat;
    background-size: cover;
    color: white;
}

.container{
    max-width: 1200px;
    margin: auto;
    width: 90%;
}

.nav {
    min-height: 8vh;
    display: flex;
    justify-content: space-between;
    align-items: center;
}

.nav-items {
    width: 40%;
    display: flex;
    list-style-type: none;
    justify-content: space-between;
}

.headline {
    text-align: center;
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    /*
    transform: translate(-50%, -50%);" is used to move
    an element (typically used for centering an element)
    to the center of its parent container both vertically
    and horizontally.

    "transform: translate(-50%, -50%);" moves an element to
    the center of its parent container by moving it 50% of
    its own size to the left and 50% of its own size up. This
    technique is often used for centering elements vertically
    and horizontally in responsive web design, where the size
    of the container may change dynamically.
    */
}

```

```
.btn {
  display: inline-block;
  outline: none;
  border: none;
  cursor: pointer;
}
```

```
.btn-headline {
  padding: 1rem 2rem;
  font-size: 1rem;
  margin-top: 1rem;
  font-weight: 600;
```

```
/*
```

*"padding: 1rem 2rem;" sets the amount of space between the content and the border of the "btn-headline" element. The "1rem" value specifies the top and bottom padding, while the "2rem" value specifies the left and right padding.*

*"font-size: 1rem;" sets the size of the text inside the "btn-headline" element to 1 rem. The "rem" unit is relative to the font-size of the root element (usually the <html> element).*

*"margin-top: 1rem;" sets the amount of space between the top edge of the "btn-headline" element and the preceding element(s). The "1rem" value specifies the amount of margin.*

*margin refers to the space outside the border of an element. It is used to add space between elements, and affects the position of the element in relation to other elements on the page. When you apply margin-top to an element, it creates space above the element and pushes it down, which affects the layout of the page.*

*the main difference between top and bottom padding vs margin-top is that padding affects the space inside the element, while margin affects the space outside the element and the position of the element in relation to other elements on the page.*

*"font-weight": In CSS, the "font-weight" property is used to set the weight (or thickness) of the font. A font's weight is typically expressed as a numerical value ranging from 100 to 900, with certain values (e.g. 100, 400, 700) having specific names (e.g. "thin", "normal", "bold").*

*A "font-weight" value of 500 means that the font is in the middle of the normal range, which is typically equivalent to "medium" or "semi-bold". It is thicker than "normal" (400) but lighter than "bold" (700). Some fonts may have additional weight values (e.g. 600, 800) that fall between these ranges.*

```
*/
```

```
}
```

```
.section-todo {
  margin-top: 5rem;
```



```

    text-align: center;
}

.form-todo {
    min-height: 5vh;
    display: flex;
    justify-content: space-between;
    margin-top: 1rem;
}

.form-todo input {
    min-height: 100%;
}

.form-todo input[type="text"] {
    width: 68%;
    padding: 0.8rem;
    font-size: 1rem;
    font-weight: 400;
}

.form-todo input[type="submit"] {
    width: 20%;
    background: rgb(44, 55, 63);
    color: white;
    font-weight: bold;
}

.section-signup {
    /*
    The auto value for the left and right margins centers the element horizontally
    within its containing block.

    So, margin: 5rem auto; centers an element horizontally and sets a large top and
    bottom margin of 5 rems each.
    */
    margin: 5rem auto;
    text-align: center;
    background: rgb(235, 232, 232);
    border-radius: 10px;
    padding: 1rem;
}

.signup-form {
    max-width: 800px;
    width: 95%;
    text-align: left;
    margin: auto;
}

```

```

}

.signup-form label {
  display: block;
}

.signup-form input {
  display: block;
  width: 100%;
  padding: 0.5rem;
}

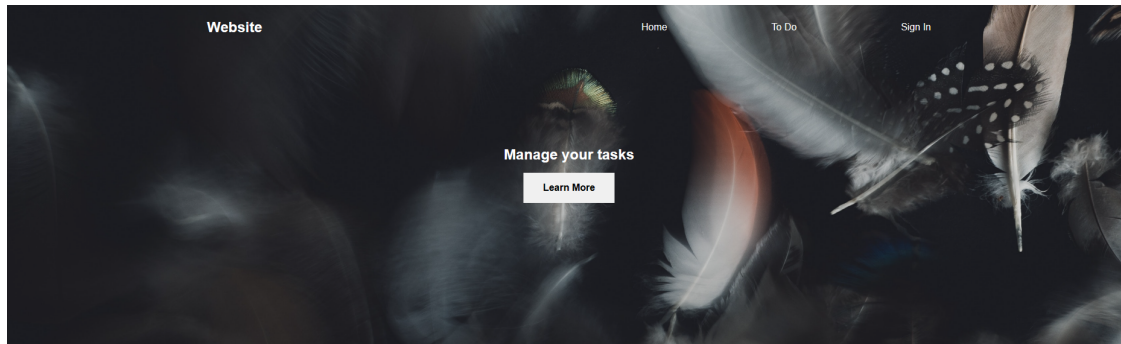
.form-group {
  margin-top: 1rem;
}

.signup-form textarea {
  width: 100%;
}

.signup-btn {
  background: rgb(44, 55, 63);
  color: white;
  padding: 1rem 2rem;
  display: block;
  margin: auto;
  margin-top: 1rem;
}

```

The HTML page looks like below:



What do you plan to do today?

Add Todo

### Sign Up

Username

password

Confirm Password

Email

About Yourself

Submit

Now, inside 102.js, if the code is only this:

```
console.log(window.document);
```

The output will be the code of this page itself:

▼ #document
[102.js:2](#)

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="./style.css">
    <!--
      We can link our js file here using below:
      <script src="./102.js"></script>
      But this will throw error if the JS is using the HTML elements below
      as the HTML elements used in this file below are not read or loaded yet

    -->
    <!--
      1st approach:
      <script src="./102.js"></script>
    -->
    <!--
      3rd approach:
      <script src="./102.js" async></script>
    -->
    <!--
      4th approach: best approach using defer
      <script src="./102.js" defer></script>
    -->
    <script src="./102.js" defer></script>
    <title>My Website</title>
  </head>
  <body>
    <header class="header"></header>
    <section class="section-todo container"></section>
    <section class="section-signup container"></section>
    <!--
      2nd approach: Putting before the end of <body> tag
      <script src="./102.js"></script>
    -->
    <!-- Code injected by live-server -->
    <script></script>
  </body>
</html>

```

In JavaScript, `console.dir()` is a method used to log an object to the console, displaying its properties in a hierarchical format, similar to expanding folders in a file system.

In the code `console.dir(window.document)`, we are using the `console.dir()` method to log the `document` object of the `window` object to the console, displaying its properties in a hierarchical format.


The `window` object is a global object in the browser environment that represents the current browser window or tab, while the `document` property of the `window` object represents the current HTML document that is loaded in the browser window.

By using `console.dir()` to log the document object, we can explore its properties and methods in a more detailed way, since `console.dir()` displays the object in a tree-like structure, allowing us to easily see its nested properties and methods.

This code logs the document object to the console using `console.dir()`, which can be helpful for debugging and understanding the structure of the HTML document and how it can be manipulated through the document object's properties and methods.

```
console.dir(window.document)
```

outputs the following:



```
▼ #document 1
  ▶ location: Location {ancestorOrigins: DOMStringList, href: 'http://127.0.0.1:5501/index.html', or
    URL: "http://127.0.0.1:5501/index.html"}
  ▶ activeElement: body
  ▶ adoptedStyleSheets: Proxy(Array) {}
  ▶ alinkColor: ""
  ▶ all: HTMLAllCollection(47) [html, head, meta, meta, meta, link, script, title, body, header.head
  ▶ anchors: HTMLCollection []
  ▶ applets: HTMLCollection []
  ▶ baseURI: "http://127.0.0.1:5501/index.html"
  ▶ bgColor: ""
  ▶ body: body
    characterSet: "UTF-8"
    charset: "UTF-8"
    childElementCount: 1
  ▶ childNodes: NodeList(2) [<!DOCTYPE html>, html]
  ▶ children: HTMLCollection [html]
    compatMode: "CSS1Compat"
    contentType: "text/html"
    cookie: ""
    currentScript: null
  ▶ defaultView: Window {window: Window, self: Window, document: document, name: '', location: Local
    designMode: "off"
    dir: ""
  ▶ doctype: <!DOCTYPE html>
  ▶ documentElement: html
    documentURI: "http://127.0.0.1:5501/index.html"
    domain: "127.0.0.1"
  ▶ embeds: HTMLCollection []
  ▶ featurePolicy: FeaturePolicy {}
```

```
console.dir(document)
```

this will also give the same output.

Same Id can't be assigned to multiple HTML elements. But classes can be assigned to multiple items.

```
// selects Id
const mainHeading = document.querySelector("#main-heading");
console.log(mainHeading);

// selects class
const header = document.querySelector(".header");
console.log(header);

// if there are multiple elements with the same class,
// it will select the first one only
const navItem = document.querySelector(".nav-item");
console.log(navItem);

// to select all elements with the same class
const navItems = document.querySelectorAll(".nav-item");
console.log(navItems);
```

```

// change text
// textContent and innerText

// print the text inside an element using Id
const mainHeading = document.getElementById("main-heading");
console.log(mainHeading.textContent)

// change the text inside an element using Id
// mainHeading.textContent = "This is changed now";
console.log(mainHeading.textContent);

// innerText will not give text inside span
console.log(mainHeading.innerText)

```

get multiple elements using `getElementsByClassName` – returns `HTMLCollection` which is array-like object

get multiple elements using `querySelectorAll` – returns `NodeList`

```

// the below gives HTMLCollection
const navItems = document.getElementsByClassName("nav-item");
console.log(navItems);

console.log(navItems[0]);

console.log(typeof navItems); // it's array-like object

console.log(Array.isArray(navItems)); // false

// returns NodeList
const navItems2 = document.querySelectorAll(".nav-item");
console.log(navItems2[1])

```

`HTMLCollection` is array-like object – can use index and has `length` property `document.getElementsByTagName` returns `HTMLCollection` or array-like object.

```

const navItems = document.getElementsByTagName("a");
console.log(navItems);
console.log(navItems.length);

```

With `HTMLCollection`, we can use simple for loop, for of loop but not `forEach` loop.

simple for loop

```

// simple for loop
for (let i = 0; i < navItems.length; i++) {
  console.log(navItems[i]);
}

```

```

for (let i = 0; i < navItems.length; i++) {
  const navItem = navItems[i];
  navItem.style.backgroundColor = "white";
  navItem.style.color = "green";
  navItem.style.fontWeight = "bold";
}

```

for of loop

```

// for of loop
// change background color and color of the link
for (let navItem of navItems){
  navItem.style.backgroundColor = "blue";
  navItem.style.color = "black";
  navItem.style.fontWeight = "bold";
}

```

forEach – will not work

```

// this will not work, will give error
navItems.forEach((navItem)=>{
  navItem.style.backgroundColor = "blue";
  navItem.style.color = "black";
  navItem.style.fontWeight = "bold";
})

```

But, we can still work with forEach loop by converting the HTMLCollection to Array using Array.from

```

// the below works
const navItemsArray = Array.from(navItems);
console.log(navItemsArray);
// Now, we can use forEach
navItemsArray.forEach((navItem)=>{
  navItem.style.backgroundColor = "azure";
  navItem.style.color = "pink";
  navItem.style.fontWeight = "bold";
})

```

## 1.0.9 querySelectorAll

**querySelectorAll** returns **NodeList** With this we can use all three loops: 1. simple for loop  
2. for of loop 3. forEach loop

```

const navItems3 = document.querySelectorAll("a");

// simple for loop
for (let i = 0; i < navItems3.length; i++) {
  const navItem = navItems3[i];
  navItem.style.backgroundColor = "white";
}

```

```

    navItem.style.color = "green";
    navItem.style.fontWeight = "bold";
}

// for of loop
for (let navItem of navItems3){
    navItem.style.backgroundColor = "blue";
    navItem.style.color = "black";
    navItem.style.fontWeight = "bold";
}

// forEach loop
navItems3.forEach((navItem)=>{
    navItem.style.backgroundColor = "azure";
    navItem.style.color = "pink";
    navItem.style.fontWeight = "bold";
})

```

NodeList can also be converted to Array using `Array.from()`

### 1.0.10 More on DOM

This is the HTML code we are working with now (`index2.html`):

```

<html>
  <head>
    <title>DOM traversing</title>
    <script src="./111.js" defer></script>
  </head>
  <body>
    <div class="container">
      <h1>My heading</h1>
      <p>Some useful information</p>
    </div>
  </body>
</html>

const rootNode = document.getRootNode();
console.log(rootNode);

console.log(rootNode.childNodes);

```



```
▼ #document 111.js:3
  <html>
    ▶ <head> ... </head>
    ▶ <body> ... </body>
  </html>

▼ NodeList [html] 1 111.js:5
  ▶ 0: html
    length: 1
  ▶ [[Prototype]]: NodeList
>
```

### 1.0.11 Root element corresponding to <HTML> tag

```
const htmlElementNode = rootNode.childNodes[0];
console.log(htmlElementNode);
```

```
<html> 111.js:8
  ▶ <head> ... </head>
  ▶ <body> ... </body>
</html>
```

If we want object representation, we can write `console.dir(htmlElementNode)`.

```
console.log(htmlElementNode.childNodes);
```

```
▼ NodeList(3) [head, text, body] 1 111.js:10
  ▶ 0: head
  ▶ 1: text
  ▶ 2: body
    length: 3
  ▶ [[Prototype]]: NodeList
```

### 1.0.12 Child nodes under root element or HTML element or HTML node

```
const headElementNode = htmlElementNode.childNodes[0];
console.log(headElementNode);
```

```
const textNode = htmlElementNode.childNodes[1];
console.log(textNode);
```

```
const bodyElementNode = htmlElementNode.childNodes[2];
console.log(bodyElementNode);
```

```
▼ <head>
  <title>DOM traversing</title>
  <script src="./111.js" defer></script>
</head>

▶ #text

▼ <body>
  <div class="container">...</div>
  <!-- Code injected by live-server -->
  <script>...</script>
</body>
```

### 1.0.13 Parent node of head element

```
console.log(headElementNode.parentNode);
```

```
<html>
  <head>...</head>
  <body>...</body>
</html>
```

[111.js:22](#)

### 1.0.14 Sibling relation

```
console.log(headElementNode.nextSibling);
```

```
▶ #text
```

[111.js:25](#)

>

### 1.0.15 Two siblings next to head element

```
// two siblings next to head element
console.log(headElementNode.nextSibling.nextSibling);
```

```
▶ <body>...</body>
```

>

### 1.0.16 nextElementSibling – it will ignore textNode consisting of space and new line

The line below will return body element:

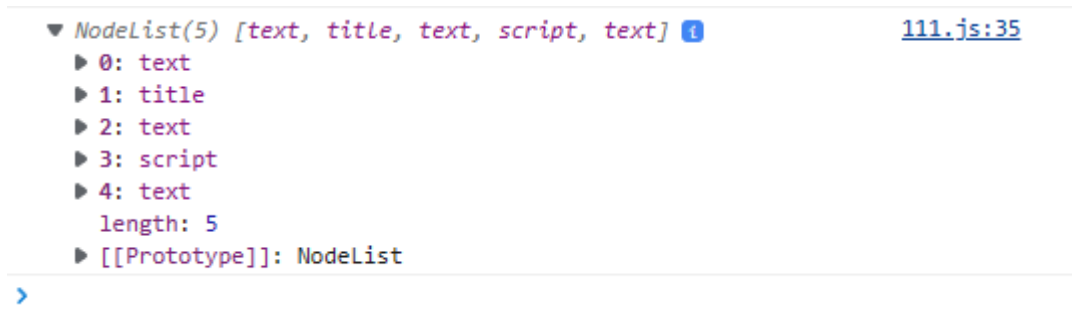
```
console.log(headElementNode.nextElementSibling);
```

```
▶ <body>...</body>
```

[111.js:33](#)

### 1.0.17 childNodes

```
console.log(headElementNode.childNodes);
```



**1.0.18** Select heading, go to its parent and then change the background color and text color of this parent

```
// Select heading, go to its parent and then change the  
// background color and text color of this parent
```

```
const h1 = document.querySelector("h1");  
const div = h1.parentNode;  
div.style.color = "#efefef";  
div.style.backgroundColor = "#333";
```

**1.0.19** Reach body element from heading and then change the background color and text color of this parent

```
// reach body element from heading and then change the  
// background color and text color of this parent
```

```
const h1 = document.querySelector("h1");  
const body = h1.parentNode.parentNode;  
body.style.color = "#efefef";  
body.style.backgroundColor = "#333";
```

**1.0.20** Direct way to select body element using `document.body`

```
// A different way to select body element and do the same tasks  
// as before
```

```
const body2 = document.body;  
body2.style.color = "#efefef";  
body2.style.backgroundColor = "#333";
```

**1.0.21** Nesting `querySelector` – selecting nested HTML element using `querySelector`

We can use `querySelector` to select an HTML element. We can again use `querySelector` to select HTML elements within the first selected element.

**Selecting title element nested inside head element** In the example below, we select head element first using `querySelector` and then select title element by again using `querySelector` on the selected head element.

```
const head = document.querySelector("head");
const title = head.querySelector("title");
console.log(title);
```

### 1.0.22 childNodes property gives child nodes including text nodes

```
// .childNodes property includes text nodes
console.log(container.childNodes); // includes text nodes
// representing new line and space
```

```
► NodeList(5) [text, h1, text, p, text]
```

[111.js:73](#)

### 1.0.23 children property gives child nodes without text nodes

```
// .children property does not include text nodes
console.log(container.children); // does not include text nodes
// representing new line and space
```

```
► HTMLCollection(2) [h1, p]
```

[111.js:78](#)

Now, again we will work with index.html file defined as following:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="./style.css">
  <!--
    We can link our js file here using below:
    <script src="./102.js"></script>
    But this will throw error if the JS is using the HTML elements below
    as the HTML elements used in this file below are not read or loaded yet

-->
  <!--
    1st approach:
    <script src="./102.js"></script>
-->
  <!--
    3rd approach:
    <script src="./102.js" async></script>
-->
  <!--
    4th approach: best approach using defer
    <script src="./102.js" defer></script>
-->
```

```

<script src="./118.js" defer></script>

<title>My Website</title>
</head>
<body>
  <header class="header">
    <nav class="nav container">
      <h1 class="log">Website</h1>
      <ul class="nav-items">
        <li class="nav-item"><a href="Home">Home</a></li>
        <li class="nav-item"><a href="">To Do</a></li>
        <li class="nav-item"><a href="">Sign In</a></li>
      </ul>
    </nav>
    <div class="headline">
      <h2 id="main-heading">Manage your tasks <span style="display: none">Hello</span></h2>
      <button class="btn btn-headline">Learn More</button>
    </div>
  </header>
  <section class="section-todo container">
    <h2>What do you plan to do today?</h2>
    <form class="form-todo">
      <input type="text" name="" id="" placeholder="Add Todo">
      <input type="submit" value="Add Todo" class="btn">
    </form>
    <ul class="todo-list">
      <!-- Uncomment the below line upto file 117.js -->
      <!--<li class="first-todo">todo 1</li> -->
      <li>item 1</li>
      <li>item 2</li>
      <li>item 3</li>
      <li>item 4</li>
      <li>item 5</li>
    </ul>
  </section>

  <section class="section-signup container">
    <h2>Sign Up</h2>
    <form class="signup-form">
      <div class="form-group">
        <label for="username">Username</label>
        <input type="text" name="username" id="username">
      </div>
      <div class="form-group">
        <label for="password">password</label>
        <input type="password" name="password" id="password">
      </div>
      <div class="form-group">

```

```

        <label for="confirmPassword">Confirm Password</label>
        <input type="password" name="confirmPassword" id="confirmPassword">
    </div>
    <div class="form-group">
        <label for="email">Email</label>
        <input type="email" name="email" id="email">
    </div>
    <div class="form-group">
        <label for="about">About Yourself</label>
        <textarea name="about" id="about" cols="30" rows="10"></textarea>
    </div>
    <button type="submit" class="btn signup-btn">Submit</button>

</form>
</section>
<!--
    2nd approach: Putting before the end of <body> tag
    <script src="./102.js"></script>
-->
</body>
</html>

```

We also have `style.css` as before:

```

@import url('https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,300;0,400;0,500;600;
*{
    box-sizing: border-box;
    padding: 0;
    margin: 0;
}

body{
    font-family: 'Poppins', sans-serif;
}

a{
    text-decoration: none;
    color: white;
}

.header{
    position: relative;
    min-height: 60vh;
    background: url('./background.jpg');
    background-position: center;
    background-repeat: no-repeat;
    background-size: cover;
    color: white;
}

```

```

}

.container{
    max-width: 1200px;
    margin: auto;
    width: 90%;
}

.nav {
    min-height: 8vh;
    display: flex;
    justify-content: space-between;
    align-items: center;
}

.nav-items {
    width: 40%;
    display: flex;
    list-style-type: none;
    justify-content: space-between;
}

.headline {
    text-align: center;
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    /*
    transform: translate(-50%, -50%);" is used to move
    an element (typically used for centering an element)
    to the center of its parent container both vertically
    and horizontally.

    "transform: translate(-50%, -50%);" moves an element to
    the center of its parent container by moving it 50% of
    its own size to the left and 50% of its own size up. This
    technique is often used for centering elements vertically
    and horizontally in responsive web design, where the size
    of the container may change dynamically.
    */
}

.btn {
    display: inline-block;
    outline: none;
    border: none;
    cursor: pointer;
}

```

```

}

.btn-headline {
  padding: 1rem 2rem;
  font-size: 1rem;
  margin-top: 1rem;
  font-weight: 600;
  /*
    "padding: 1rem 2rem;" sets the amount of space between the content and the
    border of the "btn-headline" element. The "1rem" value specifies the top and
    bottom padding, while the "2rem" value specifies the left and right padding.

    "font-size: 1rem;" sets the size of the text inside the "btn-headline" element
    to 1 rem. The "rem" unit is relative to the font-size of the root element
    (usually the <html> element).

    "margin-top: 1rem;" sets the amount of space between the top edge of the
    "btn-headline" element and the preceding element(s). The "1rem" value specifies
    the amount of margin.

    margin refers to the space outside the border of an element. It is used to add
    space between elements, and affects the position of the element in relation to
    other elements on the page. When you apply margin-top to an element, it creates
    space above the element and pushes it down, which affects the layout of the page.

    the main difference between top and bottom padding vs margin-top is that padding
    affects the space inside the element, while margin affects the space outside the
    element and the position of the element in relation to other elements on the page.

    "font-weight": In CSS, the "font-weight" property is used to set the weight
    (or thickness) of the font. A font's weight is typically expressed as a numerical
    value ranging from 100 to 900, with certain values (e.g. 100, 400, 700) having
    specific names (e.g. "thin", "normal", "bold").

    A "font-weight" value of 500 means that the font is in the middle of the normal
    range, which is typically equivalent to "medium" or "semi-bold". It is thicker
    than "normal" (400) but lighter than "bold" (700). Some fonts may have additional
    weight values (e.g. 600, 800) that fall between these ranges.
  */
}

.section-todo {
  margin-top: 5rem;
  text-align: center;
}

.bg-dark {
  background: #000;
}

```



```

    color: #eee;
}

.form-todo {
  min-height: 5vh;
  display: flex;
  justify-content: space-between;
  margin-top: 1rem;
}

.form-todo input {
  min-height: 100%;
}

.form-todo input[type="text"] {
  width: 68%;
  padding: 0.8rem;
  font-size: 1rem;
  font-weight: 400;
}

.form-todo input[type="submit"] {
  width: 20%;
  background: rgb(44, 55, 63);
  color: white;
  font-weight: bold;
}

.section-signup {
  /*
   The auto value for the left and right margins centers the element horizontally
   within its containing block.

   So, margin: 5rem auto; centers an element horizontally and sets a large top and
   bottom margin of 5 rems each.
  */
  margin: 5rem auto;
  text-align: center;
  background: rgb(235, 232, 232);
  border-radius: 10px;
  padding: 1rem;
}

.signup-form {
  max-width: 800px;
  width: 95%;
  text-align: left;
  margin: auto;
}

```

```

}

.signup-form label {
  display: block;
}

.signup-form input {
  display: block;
  width: 100%;
  padding: 0.5rem;
}

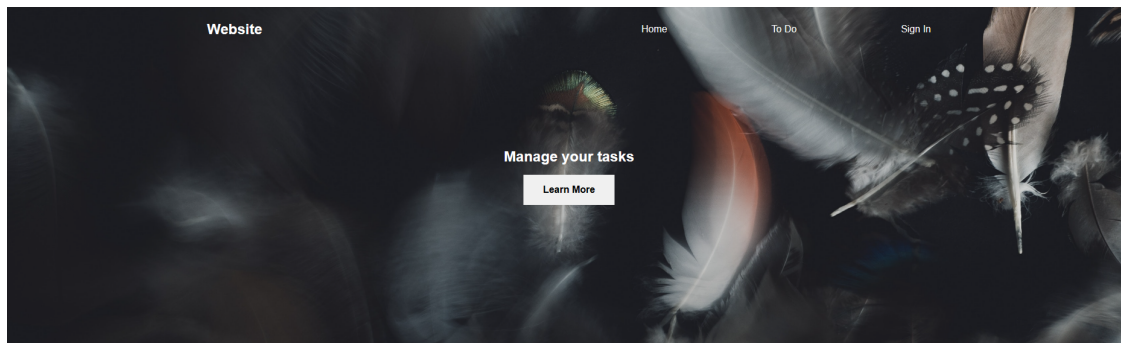
.form-group {
  margin-top: 1rem;
}

.signup-form textarea {
  width: 100%;
}

.signup-btn {
  background: rgb(44, 55, 63);
  color: white;
  padding: 1rem 2rem;
  display: block;
  margin: auto;
  margin-top: 1rem;
}

```

To refresh the memory, the HTML page looks like below:



What do you plan to do today?

### Sign Up

Username

password

Confirm Password

Email

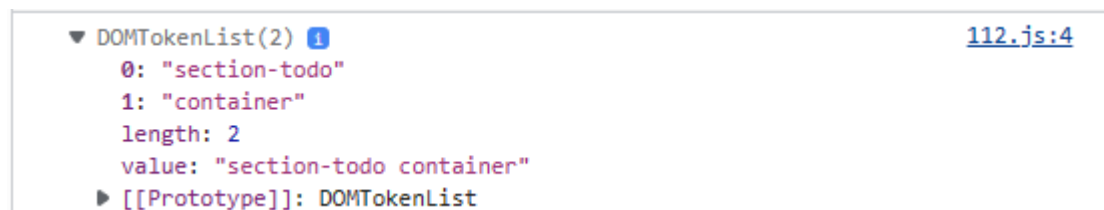
About Yourself

### 1.0.24 Listing class

```
const sectionToDo = document.querySelector('.section-todo');
```

```
// listing class
```

```
console.log(sectionToDo.classList);
```



### 1.0.25 Adding class

```
sectionToDo.classList.add('bg-dark');
```

```
console.log(sectionToDo.classList);
```



### 1.0.26 Removing class

```
sectionToDo.classList.remove("container");
console.log(sectionToDo.classList);
```

```
112.js:12
▶ DOMTokenList(2) ['section-todo', 'bg-dark', value: 'section-todo bg-dark']
>
```

### 1.0.27 Checking the existence of a class

```
console.log(sectionToDo.classList.contains('container'));
```

```
false 112.js:15
```

### 1.0.28 Toggle class

If a class is already there, remove it; if not, add it In the example, we are working with `bg-dark` class defined in CSS

```
sectionToDo.classList.toggle('bg-dark');
```

Some more code on the already covered topics.

```
const header = document.querySelector('.header');
console.log(header.classList);
header.classList.add("bg-dark");
```

### 1.0.29 Emnet abbreviation

Writing `ul.todo-list>li{todo 1}*1` in VSCode and then pressing Tab will give the following HTML code:

```
<ul class="todo-list">
  <li>todo 1</li>
</ul>
```

## 1.1 Add new HTML elements to page

### 1.1.1 Add HTML element using innerHTML

We want to select the class in this section:

```
<ul class="todo-list">
  <li>todo 1</li>
</ul>
```

The JS code to select the innerHTML inside the class `todo-list`

```
const todoList = document.querySelector(".todo-list");
console.log(todoList.innerHTML);
```

```
<li>todo 1</li>
```

[113.js:6](#)

### 1.1.2 Change the innerHTML – if we are not adding new element, we can use it

```
todoList.innerHTML = "<li>New Todo</li>"  
console.log(todoList.innerHTML);
```

```
<li>New Todo</li>
```

[113.js:10](#)

### 1.1.3 Add to the existing HTML – don't use this approach

```
todoList.innerHTML += "<li>New Todo</li>";  
console.log(todoList.innerHTML);
```

```
<li>todo 1</li>  
<li>New Todo</li>
```

[113.js:14](#)

```
// Add to the existing HTML
```

```
todoList.innerHTML += "<li>New Todo</li>";  
todoList.innerHTML += "<li>Teaching ML</li>";  
todoList.innerHTML += "<li>Learning DS and Algo</li>";  
console.log(todoList.innerHTML);
```

```
<li>todo 1</li>  
<li>New Todo</li><li>Teaching ML</li><li>Learning DS and Algo</li>
```

[113.js:16](#)

But never do what we have shown above. It gives rise to performance issues. Only use when we need to change all the code inside innerHTML, then use it. Don't use for adding new elements.

**To create new element, a better approach is using createElement**

### 1.1.4 Create HTML element using document.createElement

Some of the steps are:

- document.createElement
- append
- prepend
- remove

**Adding new list item** Let's have a look at the list items before appending anything new.

```
// todo-list before appending  
const todoList = document.querySelector(".todo-list");  
console.log(todoList);
```

```
▼ <ul class="todo-list"> 114.js:4
  <li>todo 1</li>
</ul>
```

Now, let's create new list element, create a new text node, add text to it and then add this to the list item.

```
const newTodoItem = document.createElement("li");
const newTodoItemNext = document.createTextNode("Teach ML");
newTodoItem.append(newTodoItemNext);
```

Now, add this new list item to the todo list item:

```
const newTodoItem = document.createElement("li");
const newTodoItemNext = document.createTextNode("Teach ML");
newTodoItem.append(newTodoItemNext); // Add text node to the new list node
todoList.append(newTodoItem); // add new list to the todo list
```

Now, have a look at the newly created list and the updated todo list:

```
console.log(newTodoItem);
console.log(todoList);
```

```
<li>Teach ML</li> 114.js:14
▼ <ul class="todo-list"> 114.js:15
  <li>todo 1</li>
  <li>Teach ML</li>
</ul>
```

### 1.1.5 A more efficient way to add list element – textContent

Using `textContent`, we don't even need to create a new text node (using `createTextNode`), and then add it to the newly created list node; rather, we can add text to the newly created list node using `textContent` property of this newly created list.

```
// Adding text to new list item in a more efficient way
const todoList = document.querySelector(".todo-list");
const newTodoItem = document.createElement("li");
newTodoItem.textContent = "Teach ML";
todoList.append(newTodoItem);
// we can also use appendChild() instead of append()
console.log(todoList);
```

```
▼ <ul class="todo-list"> 114.2.js:7
  <li>todo 1</li>
  <li>Teach ML</li>
</ul>
```

Now, we also look at `prepend` which adds at the beginning:

```
// Adding text to new list item in a more efficient way
const todoList = document.querySelector(".todo-list");
```

```
const newTodoItem = document.createElement("li");
newTodoItem.textContent = "Teach ML";
todoList.prepend(newTodoItem);
console.log(todoList);
```

```
▼ <ul class="todo-list">                                     114 3.js:6
  <li>Teach ML</li>
  <li>todo 1</li>
</ul>
```

### 1.1.6 Removing a list item

```
// selects first list item under todo-list class
const todo1 = document.querySelector(".todo-list li");
todo1.remove();
console.log(todo1);
```

### 1.1.7 Inserting before and after an element

Inserting before using before()

```
// Inserting before
const newTodoItem = document.createElement("li");
newTodoItem.textContent = "Teach ML";
const todoList = document.querySelector(".todo-list");
todoList.before(newTodoItem);
```

Inserting after using after()

```
// Inserting after
const newTodoItem = document.createElement("li");
newTodoItem.textContent = "Teach ML";
const todoList = document.querySelector(".todo-list");
todoList.after(newTodoItem);
```

### 1.1.8 An alternative way to insert element using insertAdjacentHTML

- beforebegin
- afterbegin
- beforeend
- afterend

`elem.insertAdjacentHTML(where, html)` Insert before the end of the element with class `todo-list`

```
// Add one more list item under unordered list
const todoList = document.querySelector(".todo-list");
console.log(todoList);
```

```

todoList.insertAdjacentHTML("beforeend", "<li>Teach ML</li>");
console.log(todoList);

```

Similarly, `afterbegin` can be used for prepending

```

todoList.insertAdjacentHTML("afterbegin", "<li>Added using afterbegin</li>");
console.log(todoList);

```

Similarly, we can use `beforebegin` and `afterend`

```

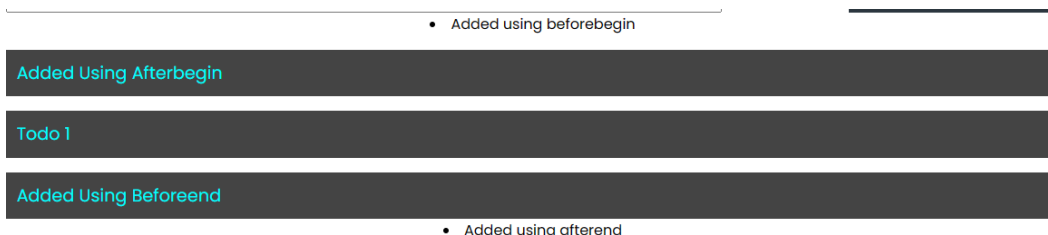
todoList.insertAdjacentHTML("beforebegin", "<li>Added using beforebegin</li>");
console.log(todoList);

```

```

// It will be added outside the unordered list or class with the value todo-list
todoList.insertAdjacentHTML("afterend", "<li>Added using afterend</li>");
console.log(todoList);

```



### 1.1.9 Clone nodes

Suppose we want to both append and prepend a new item to the list, we can clone and then append/prepend the cloned node

```

const ul = document.querySelector(".todo-list");
const li = document.createElement("li");
li.textContent = "Teach ML";
const li2 = li.cloneNode(true); // true means deep clone
ul.prepend(li);
ul.append(li2);

```

### 1.1.10 Some old methods to support poor IE

- `appendChild` (instead of `append`)

```

const ul = document.querySelector(".todo-list");
const li = document.createElement("li");
li.textContent = "New todo";
ul.appendChild(li);

```

- `insertBefore` (instead of `prepend`)

```

// use of insertBefore
const ul = document.querySelector(".todo-list");
const li = document.createElement("li");
const referenceNode = document.querySelector(".first-node");

```



```
li.textContent = "Added with insertBefore";
ul.insertBefore(li, referenceNode);

• replaceChild;

const ul = document.querySelector(".todo-list");
const li = document.createElement("li");
li.textContent = "Added with insertBefore";
const referenceNode = document.querySelector(".first-node");
ul.replaceChild(li, referenceNode);

• removeChild (instead of remove)

const ul = document.querySelector(".todo-list");
const referenceNode = document.querySelector(".first-node");
ul.removeChild(referenceNode);
```

### 1.1.11 More Emnet shortcut

li{item \$}\*5 will give

```
<li>item 1</li>
<li>item 2</li>
<li>item 3</li>
<li>item 4</li>
<li>item 5</li>
```

### 1.1.12 static list vs live list

querySelectorAll gives us static list, gives us NodeList

getElementBySomething gives us live list, gives us HTML Element

static list

```
const listItems = document.querySelectorAll(".todo-list li");
const sixthListItem = document.createElement("li");
sixthListItem.textContent = "Sixth todo -- added with static list";
const ul = document.querySelector(".todo-list");
ul.append(sixthListItem);
console.log(listItems); // this will not include the new item
// although the new item is added to the DOM, the static
// list is not updated
```

live list

```
const ul = document.querySelector(".todo-list");
const listItems = ul.getElementsByTagName("li");

const sixthListItem = document.createElement("li");
sixthListItem.textContent = "Sixth todo -- added with live list";
const ul = document.querySelector(".todo-list");
```

```
ul.append(sixthListItem);
console.log(listItems); // this will include the new item
```

### 1.1.13 Dimensions of an element

#### getBoundingClientRect()

```
const sectionTodo = document.querySelector('.section-todo');
const info = sectionTodo.getBoundingClientRect();
console.log(info);
```

```
DOMRect {x: 64.453125, y: 365.09375, width: 1160.09375, height: 415.46875,
  top: 365.09375, ...}
  bottom: 780.5625
  height: 415.46875
  left: 64.453125
  right: 1224.546875
  top: 365.09375
  width: 1160.09375
  x: 64.453125
  y: 365.09375
  [[Prototype]]: DOMRect
```

```
const height = sectionTodo.getBoundingClientRect().height;
console.log(height);
```

443.46875

[119.js:11](#)

```
const width = sectionTodo.getBoundingClientRect().width;
console.log(width);
```

217.796875

[119.js:14](#)

```
const top = sectionTodo.getBoundingClientRect().top;
console.log(top);
```

96.5

[119.js:17](#)

### 1.1.14 Events

In JavaScript, an event is a signal that is generated by the browser or by the user's interaction with a web page. Events can be triggered by various actions, such as clicking a button, submitting a form, moving the mouse, or typing on the keyboard.

When an event occurs, the browser creates an event object that contains information about the event, such as its type, target element, and any additional data related to the event.

You can use JavaScript to listen for events and respond to them in different ways. This is often done by attaching event listeners to elements in the web page using the `addEventListener` method. Here's an example:

```
const button = document.querySelector('#my-button');

button.addEventListener('click', function(event) {
    // do something when the button is clicked
});
```

In this example, an event listener is attached to a button element with the ID `my-button`. The listener responds to the `click` event, which is triggered when the button is clicked. When the event occurs, the function passed as the second argument to `addEventListener` is called, and the event object is passed as an argument to the function.

Overall, events in JavaScript allow you to create dynamic and interactive web pages by responding to user interactions and other actions in the browser.

### There are 3 ways to bind events to elements

1. Using `onclick` attribute inside the element
2. Select the element and specify what to do when a specific event happens
3. Select the element and use `addEventListener` method (more efficient and flexible)

#### 1.1.15 1. using `onclick` attribute inside the HTML element inside the HTML file

```
<button class="btn btn-headline" onclick="console.log('You clicked me!')">Learn More</button>
```

#### 1.1.16 2. select the element and specify what to do when the event happens

```
const btn = document.querySelector(".btn-headline");
btn.onclick = function(){
    console.log("You clicked me!");
}
```

#### 1.1.17 3. Select the element and use/add `addEventListener` method

Below all three approaches give the same result.

##### Approach 1

```
const btn = document.querySelector(".btn-headline");
function clickMe(){
    console.log("You clicked me!");
}
btn.addEventListener("click", clickMe);
```

##### Approach 2

```
const btn = document.querySelector(".btn-headline");
btn.addEventListener("click", function(){
    console.log("You clicked me!");
})
```

### Approach 3

```
const btn = document.querySelector(".btn-headline");
btn.addEventListener("click", ()=>{
  console.log("Arrow function -- You clicked me!");
});
```

#### 1.1.18 this keyword

```
const btn = document.querySelector(".btn-headline");
btn.addEventListener("click", function(){
  console.log("you clicked me");
  console.log(this);
  this.style.backgroundColor = "red";
});
```

you clicked me	<a href="#">121.js:6</a>
	<a href="#">121.js:7</a>
<pre>&lt;button class="btn btn-headline" style="background-color: red;"&gt;Learn More &lt;/button&gt;</pre>	
>	


Even if we declare function outside of the `addEventListener` method, the `this` keyword will still refer to the element that we are listening to the event on.

```
const btn = document.querySelector(".btn-headline");
function clickMe(){
  console.log("you clicked me");
  console.log(this);
  this.style.backgroundColor = "red";
}
btn.addEventListener("click", clickMe);
```

you clicked me	<a href="#">121.js:6</a>
	<a href="#">121.js:7</a>
<pre>&lt;button class="btn btn-headline" style="background-color: red;"&gt;Learn More &lt;/button&gt;</pre>	
>	

In case of arrow function, the value of `this` keyword will be window or will be the value of `this` keyword in the parent scope

```
const btn = document.querySelector(".btn-headline");
btn.addEventListener("click", ()=>{
  console.log("you clicked me");
  console.log(this);
  //this.style.backgroundColor = "red";
});
```

you clicked me	<a href="#">121.js:33</a>
 <pre>Window {window: Window, self: Window, document: document, name: '', location: Location, ...}</pre>	<a href="#">121.js:34</a>
>	

### 1.1.19 More Emmet shortcuts

**! + Tab** **! + Tab** gives the following HTML template code

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
</body>
</html>
```

**script:src + Tab** **script:src + Tab** gives the following script tag:

```
<script src=""></script>
```

**.className + Tab** **.className + Tab** gives the following div with the class `className`

For example, **.mybuttons + Tab** gives the following div

```
<div class="mybuttons">
```

**button\*3 + Tab** **\*\*button\*3 + Tab\*\*** gives 3 button elements

```
<button></button><button></button><button></button>
```

**h1.heading\$\*7{Hello World}** **\*\* h1.heading\$\*7{Hello World}\*\*** will give 7 h1 with class heading and text Hello World

```
<h1 class="heading1">Hello World</h1>
<h1 class="heading2">Hello World</h1>
<h1 class="heading3">Hello World</h1>
<h1 class="heading4">Hello World</h1>
<h1 class="heading5">Hello World</h1>
<h1 class="heading6">Hello World</h1>
<h1 class="heading7">Hello World</h1>
```

### 1.1.20 Adding click event to multiple elements (all the buttons)

I have the following HTML file and trying to incorporate click events to all the buttons

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="./122.js" defer></script>
  <title>Click event</title>
</head>
<body>
  <div class="my-buttons">
    <button id="one">My button one</button>
    <button id="two">My button two</button>
    <button id="three">My button three</button>
  </div>
</body>
</html>

```

Now, in the **122.js** file, we can write the loops in different ways to add click events to multiple elements

### 1. Using simple for loop

```

const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

for(let i=0; i<allButtons.length; i++){
  allButtons[i].addEventListener("click", function(){
    console.log("You clicked me!");
    console.log(this); // will show the button element
  })
}

```

Clicking on the first button will show this:

You clicked me!	<a href="#">122.js:12</a>
<u>&lt;button id="one"&gt;My button one&lt;/button&gt;</u>	<a href="#">122.js:13</a>

Clicking on the second button will show this:

You clicked me!	<a href="#">122.js:12</a>
<u>&lt;button id="two"&gt;My button two&lt;/button&gt;</u>	<a href="#">122.js:13</a>

Clicking on the third button will show this:

You clicked me!	<a href="#">122.js:12</a>
<u>&lt;button id="three"&gt;My button three&lt;/button&gt;</u>	<a href="#">122.js:13</a>

We can also see the text content of these different buttons:

```
for(let i=0; i<allButtons.length; i++){
    allButtons[i].addEventListener("click", function(){
        console.log(this.textContent); // will show the button text
    })
}
```

Clicking on the second button will now show the text content of this button:



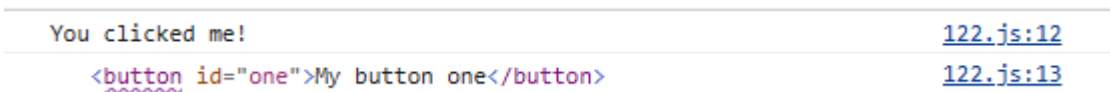
**Note:** Don't use arrow function with the above code as this will refer to the window object and not the button element. But also remember that most of the time we will be using arrow functions. We will see how to get the button element using arrow functions later.

**But how do I use arrow function then – use `event.target`** If we just use arrow function here, `this` will give the window object. Using event object, we can get the same result. Event object is discussed later.

```
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

for(let i=0; i<allButtons.length; i++){
    allButtons[i].addEventListener("click", (e)=>{
        console.log("You clicked me!");
        console.log(e.target.textContent); // will show the button element
    })
}
```

Now, clicking on the first button will show this:



## 2. Using for of loop

*// Using for of loop*

```
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

for(let button of allButtons){
    button.addEventListener("click", function(){
        //console.log("You clicked me!");
        //console.log(this);
        console.log(this.textContent);
    })
}
```

Similar to before, if we click on the third button, we will see the following in the Console:

My button three	<a href="#">122.js:43</a>
>	

Using arrow function – `event.target`

```
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

for(let button of allButtons){
  button.addEventListener("click", (e)=>{
    //console.log("You clicked me!");
    //console.log(this);
    console.log(e.target.textContent);
  })
}
```

Similar to before, if we click on the third button, we will see the following in the Console:

My button three	<a href="#">122.js:43</a>
>	

### 3. Using `forEach` method

```
// using forEach method
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

allButtons.forEach(function(button){
  button.addEventListener("click", function(){
    // console.log("You clicked me!");
    // console.log(this);
    console.log(this.textContent);
  })
});
```

Similar to before, if we click on the third button, we will again see the following in the Console:

My button three	<a href="#">122.js:57</a>
>	

Using arrow function – `event.target`

```
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

allButtons.forEach(function(button){
  button.addEventListener("click", (e)=>{
    // console.log("You clicked me!");
    // console.log(this);
    console.log(e.target.textContent);
  })
});
```



```
    })  
  });
```

If clicked on button three



### 1.1.21 Event object

Whenever browser encounters an event it is listening to 1. Gives callback function to JS Engine  
2. With the callback function, it also gives information on the event happened or an object called event object

```
const firstButton = document.querySelector("#one");  
  
firstButton.addEventListener("click", function(abc){  
  console.log(abc); // will show the event object  
});
```

Now, if we click on button one with id “one”, we get the following in the Console:

123.js:24

```

▼ PointerEvent {isTrusted: true, pointerId: 1, width: 1, height: 1, pressure: 0, ...} ⓘ
  isTrusted: true
  altKey: false
  altitudeAngle: 1.5707963267948966
  azimuthAngle: 0
  bubbles: true
  button: 0
  buttons: 0
  cancelBubble: false
  cancelable: true
  clientX: 88
  clientY: 11
  composed: true
  ctrlKey: false
  currentTarget: null
  defaultPrevented: false
  detail: 1
  eventPhase: 0
  fromElement: null
  height: 1
  isPrimary: false
  layerX: 88
  layerY: 11
  metaKey: false
  movementX: 0
  movementY: 0
  offsetX: 78
  offsetY: 1
  pageX: 88
  pageY: 11
  pointerId: 1
  pointerType: "mouse"
  pressure: 0
  relatedTarget: null
  returnValue: true
  screenX: 87
  screenY: 535
  shiftKey: false
  ▶ sourceCapabilities: InputDeviceCapabilities {firesTouchEvents: false}
  ▶ srcElement: button#one
    tangentialPressure: 0
  ▶ target: button#one
    tiltX: 0

```

2 important properties of event:

1. **target** Which element triggered the event

```

▼ target: button#one
  accessKey: ""
  ariaAtomic: null
  ariaAutoComplete: null
  ariaBrailleLabel: null
  ariaBrailleRoleDescription: null
  ariaBusy: null
  ariaChecked: null
  ariaColCount: null
  ariaColIndex: null
  ariaColSpan: null
  ariaCurrent: null
  ariaDescription: null
  ariaDisabled: null
  ariaExpanded: null
  ariaHasPopup: null
  ariaHidden: null
  ariaInvalid: null
  ariaKeyShortcuts: null
  ariaLabel: null
  ariaLevel: null
  ariaLive: null
  ariaModal: null
  ariaMultiline: null
  ariaMultiSelectable: null
  ariaOrientation: null
  ariaPlaceholder: null
  ariaPosInSet: null
  ariaPressed: null
  ariaReadOnly: null
  ariaRelevant: null
  ariaRequired: null
  ariaRoleDescription: null
  ariaRowCount: null
  ariaRowIndex: null
  ariaRowSpan: null
  ariaSelected: null
  ariaSetSize: null
  ariaSort: null
  ariaValueMax: null
  ariaValueMin: null
  ariaValueNow: null
  ariaValueText: null

```

2. **currentTarget** to which element we attached the event listener

```
currentTarget: null
```

Now using `event.target` in the arrow function will address the issue of not being able to use for loop to add key events to buttons.

```

const firstButton = document.querySelector("#one");

firstButton.addEventListener("click", function(abc){

```

```
    console.log(abc.target); // will show the event object
  });
```

Clicking on the first button gives:

```
<button id="one">My button one</button> 123.js:26
```

The following code will attach click event to all the buttons

```
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

for(let button of allButtons){
  button.addEventListener("click", (e)=>{
    console.log(e.currentTarget);
  })
}
```

Now, if we click on the three buttons:

```
<button id="one">My button one</button> 123.js:46
<button id="two">My button two</button> 123.js:46
<button id="three">My button three</button> 123.js:46
>
```

For the code example we used, it doesn't matter whether we use `event.target` or `event.currentTarget`. But, later we will see where they actually differ.

JS Engine of **Chrome** is **v8**

JS Engine of Firefox is **Spider Monkey**

### 1.1.22 Event loop, callback queue

The event loop and callback queue are two important concepts in JavaScript that help manage the execution of asynchronous code.

In JavaScript, the event loop is a mechanism that ensures the code runs in a non-blocking way. The event loop continuously checks for tasks in the callback queue and executes them one by one in a sequential order. The event loop keeps running until there are no more tasks in the callback queue.

When an asynchronous operation is started in JavaScript, it is placed in the callback queue when it completes, along with any associated callback functions. These callbacks are executed in the order they are received in the callback queue.

Here's an example to illustrate the event loop and callback queue:

```
[2]: console.log('Start');
      setTimeout(() => {
        console.log('Middle');
```

```
}, 1000);  
console.log('End');
```

Start  
End  
Middle

In this example, we have a `setTimeout` function that takes a callback function as its first argument and a time delay of 1000ms as its second argument. When the `setTimeout` function is called, it starts an asynchronous operation that waits for the specified time and then adds the callback function to the callback queue.

As you can see, the `console.log` statements are executed in the order they appear in the code. However, the callback function added to the callback queue by the `setTimeout` function is executed after the main code has finished executing.

In summary, the event loop and callback queue are essential for managing the execution of asynchronous code in JavaScript. The event loop continuously checks for tasks in the callback queue and executes them one by one in a sequential order, ensuring that the code runs in a non-blocking way.

### 1.1.23 Event bubbling, capturing

Event bubbling and capturing are two mechanisms in JavaScript that describe the order in which event handlers are executed when an event occurs on an element with nested child elements.

Event capturing is the first phase of event propagation, where the event is captured by the outermost element first and then propagated inward to the target element. This means that if you have a nested structure of HTML elements, such as a div inside another div, and an event occurs on the inner div, the event will first be captured by the outer div before it is handled by the inner div. The capturing phase occurs before the bubbling phase.

Event bubbling is the second phase of event propagation, where the event is handled by the target element first and then propagated outward to the outermost element. This means that if you have a nested structure of HTML elements and an event occurs on the innermost element, the event will first be handled by the innermost element and then propagated up to the outermost element. The bubbling phase occurs after the capturing phase.

Event bubbling is the default mechanism in JavaScript, and it can be useful for handling events that are triggered by child elements. However, in some cases, you may want to use event capturing instead. To use event capturing, you can set the third parameter of the `addEventListener` method to `true`.

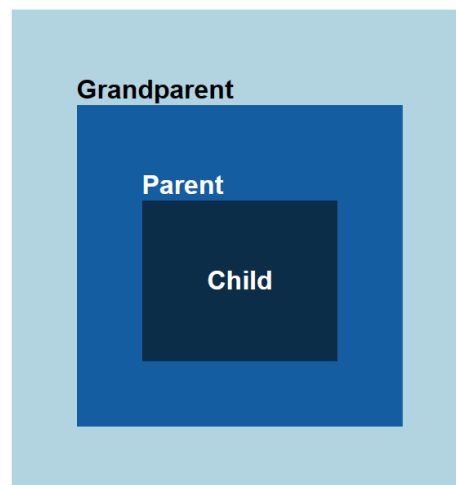
We have the following HTML file:

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <meta http-equiv="X-UA-Compatible" content="IE=edge">  
  <meta name="viewport" content="width=device-width, initial-scale=1.0">  
  <link rel="stylesheet" href="./evt-bcd.css">
```

```

<script src="./128.js" defer></script>
<title>Event Bubbling | Capturing | delegation</title>
</head>
<body>
  <main>
    <div class="grandparent box">
      Grandparent
      <div class="parent box">
        Parent
        <div class="child box">Child</div>
      </div>
    </div>
  </main>
</body>
</html>

```



The page looks like below:

The JavaScript code we are using is in **128.js**.

```

// console.log("Hello world!");

const grandparent = document.querySelector(".grandparent");
const parent = document.querySelector(".parent");
const child = document.querySelector(".child");

// capturing events
// from the top to the bottom
child.addEventListener("click", ()=> {
  console.log("Capture !!! child");
});

```

```

}, true);

parent.addEventListener("click", ()=> {
  console.log("Capture !!! parent");
}, true);

grandparent.addEventListener("click", ()=> {
  console.log("Capture !!! grandparent");
}, true);

document.body.addEventListener("click", ()=> {
  console.log("Capture !!! body");
}, true);

// not capture -- bubbling
// from the bottom to the top
child.addEventListener("click", ()=> {
  console.log("Bubble child");
});

parent.addEventListener("click", ()=> {
  console.log("Bubble parent");
});

grandparent.addEventListener("click", ()=> {
  console.log("Bubble grandparent");
});

document.body.addEventListener("click", ()=> {
  console.log("Bubble body");
});

```

Now, if we click outside the **grandparent** portion inside the body, we get the following output:

Capture !!! body	<a href="#">128.js:23</a>
Bubble body	<a href="#">128.js:42</a>
>	

Now, if we click inside the **grandparent** portion, we get the following output:

Capture !!! body	<a href="#">128.js:23</a>
Capture !!! grandparent	<a href="#">128.js:19</a>
Bubble grandparent	<a href="#">128.js:38</a>
Bubble body	<a href="#">128.js:42</a>
.	

Now, if we click inside the **parent** portion, we get the following output:

Capture !!! body	<a href="#">128.js:23</a>
Capture !!! grandparent	<a href="#">128.js:19</a>
Capture !!! parent	<a href="#">128.js:15</a>
Bubble parent	<a href="#">128.js:34</a>
Bubble grandparent	<a href="#">128.js:38</a>
Bubble body	<a href="#">128.js:42</a>

Now, if we click inside the child portion, we get the following output:

Capture !!! body	<a href="#">128.js:23</a>
Capture !!! grandparent	<a href="#">128.js:19</a>
Capture !!! parent	<a href="#">128.js:15</a>
Capture !!! child	<a href="#">128.js:11</a>
Bubble child	<a href="#">128.js:30</a>
Bubble parent	<a href="#">128.js:34</a>
Bubble grandparent	<a href="#">128.js:38</a>
Bubble body	<a href="#">128.js:42</a>

Now, let's change our js file (128\_2.js) to below where for capturing events, we keep only grandparent and body events and for bubbling, we keep only parent and child events:

```
const grandparent = document.querySelector(".grandparent");
const parent = document.querySelector(".parent");
const child = document.querySelector(".child");

// capturing events
// from the top to the bottom
grandparent.addEventListener("click", ()=> {
  console.log("Capture !!! grandparent");
}, true);

document.body.addEventListener("click", ()=> {
  console.log("Capture !!! body");
}, true);

// not capture
// from the bottom to the top
child.addEventListener("click", ()=> {
  console.log("Bubble child");
});

parent.addEventListener("click", ()=> {
  console.log("Bubble parent");
});
```

Now, if we click inside child portion, we will see this:



Capture !!! body	<a href="#">128_2.js:14</a>
Capture !!! grandparent	<a href="#">128_2.js:10</a>
Bubble child	<a href="#">128_2.js:21</a>
Bubble parent	<a href="#">128_2.js:25</a>
>	

Now, if we change the JS code (128\_3.js) as such that event is associated with only grandparent

*// event delegation*

```
const grandparent = document.querySelector(".grandparent");

grandparent.addEventListener("click", (e)=> {
  console.log("You clicked something!");
});
```

Now, even if we click inside child element which doesn't have any click event associated, we will see "You clicked something!" in the console.

You clicked something!	<a href="#">128_3.js:7</a>
------------------------	----------------------------

It starts looking for event starting from body, then goes to check grandparent, then parent, then child, then it again bubbles to see if there is any click event in parent, if not if there is any click event associated with Grandparent, then it called the callback function of the grandparent and not child. This means child, parent and grandparent don't need separate event listeners due to event delegation.

If the js code is updated (128\_4.js),

```
const grandparent = document.querySelector(".grandparent");

grandparent.addEventListener("click", (e)=> {
  console.log(e);
});
```

We will see that PointerEvent is printed and the value of target is div.child.box.

We can also print the HTML tag for the portion where it is clicked,

```
const grandparent = document.querySelector(".grandparent");

grandparent.addEventListener("click", (e)=> {
  console.log(e.target);
});
```

Now, if we click inside child portion, we will see the HTML tag for child:

<div class="child box">Child</div>	<a href="#">128_5.js:6</a>
------------------------------------	----------------------------

Similarly, if we click at the parent portion, we will see the HTML tag for parent. This will include the HTML tag for child as it is nested inside parent.

```
▼ <div class="parent box"> 128 5.js:6
  " Parent "
  <div class="child box">Child</div>
</div>
```

Similarly, if we click at the grandparent portion, we will see the HTML tag for grandparent. This will include the HTML tag for parent and child as they are nested inside grandparent.

```
128 5.js:6
▼ <div class="grandparent box">
  " Grandparent "
  ▼ <div class="parent box">
    " Parent "
    <div class="child box">Child</div>
  </div>
</div>
```

Now, if we click in the body part, nothing will happen, as we have not associated any event with body and so event capture doesn't happen.

```
const grandparent = document.querySelector(".grandparent");

grandparent.addEventListener("click", (e) => {
  console.log(e.target.textContent);
});
```

Now, clicking on parent will give "Parent Child", clicking on grandparent will give "Grandparent Parent Child" and clicking on child will only give "Child".

[ ]: