# JavaScript

# DOM tree

The backbone of an HTML document is tags.
According to the Document Object Model (DOM), every HTML tag is an object. Nested tags are “children” of the enclosing one. The text inside a tag is an object as well.
All these objects are accessible using JavaScript, and we can use them to modify the page.
For example, document.body is the object representing the <body> tag.

Running this code will make the <body> red for 3 seconds:

```
document.body.style.background = 'red'; // make the background red

setTimeout(() => document.body.style.background = '', 3000); // return back
```

# DOM tree

**On top: documentElement and body**

The topmost tree nodes are available directly as document properties:

**<html> = document.documentElement**

The topmost document node is document.documentElement. That's the DOM node of the <html> tag.

**<body> = document.body**

Another widely used DOM node is the <body> element – document.body.

**<head> = document.head**

The <head> tag is available as document.head.

# DOM tree

**Children: childNodes, firstChild, lastChild**

There are two terms that we'll use from now on:

**Child nodes (or children)** – elements that are direct children. In other words, they are nested exactly in the given one. For instance, <head> and <body> are children of <html> element.

**Descendants** – all elements that are nested in the given one, including children, their children and so on.

For instance, here <body> has children  and <ul> (and few blank text nodes):

```
<html>
<body>
  <div>Begin</div>

  <ul>
    <li>
      <b>Information</b>
    </li>
  </ul>
</body>
</html>
```

# DOM tree

**The childNodes collection lists all child nodes, including text nodes.**
The example below shows children of document.body:

```
<html>
<body>
  <div>Begin</div>

  <ul>
    <li>Information</li>
  </ul>

  <div>End</div>

  <script>
    for (let i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ..., SCRIPT
    }
  </script>
  ...more stuff...
</body>
</html>
```

# DOM tree

**Searching: getElement*, querySelector***

DOM navigation properties are great when elements are close to each other. What if they are not? How to get an arbitrary element of the page?

There are additional searching methods for that.

**document.getElementById or just id**

If an element has the id attribute, we can get the element using the method document.getElementById(id), no matter where it is.

For instance:

```
<div id="elem">
  <div id="elem-content">Element</div>
</div>

<script>
  // get the element
  let elem = document.getElementById('elem');

  // make its background red
  elem.style.background = 'red';
</script>
```

# DOM tree

**querySelectorAll**

By far, the most versatile method, elem.querySelectorAll(css) returns all elements inside elem matching the given CSS selector.

Here we look for all <li> elements that are last children:

```
<ul>
  <li>The</li>
  <li>test</li>
</ul>
<ul>
  <li>has</li>
  <li>passed</li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');

  for (let elem of elements) {
    alert(elem.innerHTML); // "test", "passed"
  }
</script>
```

# DOM tree

**getElementsBy***

There are also other methods to look for nodes by a tag, class, etc.
Today, they are mostly history, as querySelector is more powerful and shorter to write.
So here we cover them mainly for completeness, while you can still find them in the old scripts.
elem.getElementsByTagName(tag) looks for elements with the given tag and returns the collection of them. The tag parameter can also be a star "*" for “any tags”.

elem.getElementsByClassName(className) returns elements that have the given CSS class.

document.getElementsByName(name) returns elements with the given name attribute, document-wide. Very rarely used.

**For instance:**

```
// get all divs in the document
let divs = document.getElementsByTagName('div');
```

# DOM tree

**innerHTML: the contents**

The innerHTML property allows to get the HTML inside the element as a string.
We can also modify it. So it's one of the most powerful ways to change the page.
The example shows the contents of document.body and then replaces it completely:

```
<body>
  <p>A paragraph</p>
  <div>A div</div>

  <script>
    alert( document.body.innerHTML ); // read the current contents
    document.body.innerHTML = 'The new BODY!'; // replace it
  </script>

</body>
```

# DOM tree

**nodeValue/data: text node content**

The innerHTML property is only valid for element nodes.
Other node types, such as text nodes, have their counterpart: nodeValue and data properties. These two are almost the same for practical use, there are only minor specification differences. So we'll use data, because it's shorter.
An example of reading the content of a text node and a comment:

```
<body>
  Hello
  <!-- Comment -->
  <script>
    let text = document.body.firstChild;
    alert(text.data); // Hello

    let comment = text.nextSibling;
    alert(comment.data); // Comment
  </script>
</body>
```

# DOM tree

**textContent: pure text**

The textContent provides access to the *text* inside the element: only text, minus all <tags>.

For instance:

```
<div id="news">
  <h1>Headline!</h1>
  <p>Martians attack people!</p>
</div>

<script>
  // Headline! Martians attack people!
  alert(news.textContent);
</script>
```

# DOM tree

## Difference Between innerHTML vs textContent

# DOM tree

## Difference Between innerHTML vs textContent

| innerHTML | textContent |
| --- | --- |
| Reads HTML | Reads only text |
| Can insert HTML | Cannot insert HTML |
| Less safe | Safer |

# DOM tree

Compare the two:

```
<div id="elem1"></div>
<div id="elem2"></div>

<script>
  let name = prompt("What's your name?", "<b>Winnie-the-Pooh!</b>");

  elem1.innerHTML = name;
  elem2.textContent = name;
</script>
```

Output???????????????

# DOM tree

**The “hidden” property**

The “hidden” attribute and the DOM property specifies whether the element is visible or not.

We can use it in HTML or assign it using JavaScript, like this:

```
<div>Both divs below are hidden</div>

<div hidden>With the attribute "hidden"</div>

<div id="elem">JavaScript assigned the property "hidden"</div>

<script>
  elem.hidden = true;
</script>
```

Technically, hidden works the same as style="display:none". But it’s shorter to write.

# DOM tree

**More properties**

DOM elements also have additional properties, in particular those that depend on the class:

value – the value

for <input>, <select> and <textarea> (HTMLInputElement, HTMLSelectElement…).

href – the “href” for <a href="..."> (HTMLAnchorElement).

id – the value of “id” attribute, for all elements (HTMLElement).

…and much more…

For instance:

```
<input type="text" id="elem" value="value">

<script>
  alert(elem.type); // "text"
  alert(elem.id); // "elem"
  alert(elem.value); // value
</script>
```

# DOM tree

**What is an Event?**
An event is an action that happens in the browser:
Click
Submit
Key press
Mouse move
Load
**Why Events Matter?**
They make websites interactive.
**Ways to Handle Events**
**--Inline Event (Not Recommended)**
<button onclick="sayHello()">Click</button>
**--DOM Property**
btn.onclick = function() {};
--**addEventListener (Recommended ✅)**
btn.addEventListener("click", function() {});

# DOM tree

**The Event Object**
**What is Event Object?**
It contains information about the event.

```
btn.addEventListener("click", function(event) {
   console.log(event.target);
});
```

Important properties:
event.target
event.type
event.preventDefault()

**Prevent Default Behavior**
Used mainly with forms.

```
form.addEventListener("submit", function(e) {
   e.preventDefault();
});
```

Why?
To stop page refresh.

# DOM tree

**Common DOM Events**
Mouse Events:
click
dblclick
mouseover
mouseout
Keyboard Events:
keydown
keyup
Form Events:
submit
change
input

# DOM tree

**DOM Manipulation Advanced**
**Creating Elements**
**createElement()**
let div = document.createElement("div");
**appendChild()**
document.body.appendChild(div);

**Adding & Removing Elements**
**prepend()**
Adds element at beginning
**remove()**
Removes element
element.remove();

**Working with classList**
**classList Methods:**
add()
remove()
toggle()
contains()
Example:  element.classList.toggle("active");

# DOM tree

**Working with Attributes**
**Attribute Methods:**
getAttribute()
setAttribute()
removeAttribute()
hasAttribute()
Example:
input.setAttribute("placeholder", "Enter name");

# DOM tree

**Forms & Validation**
**Forms with JavaScript**
Steps:
Select form
Listen to submit
Prevent default
Validate inputs

**— Reading Input Values**
let value = input.value;
Important:
trim()
length

**— Displaying Errors in DOM**
Use:
textContent
classList
style
Example:
errorMessage.textContent = "Password too short";

# DOM tree

**DOMContentLoaded**

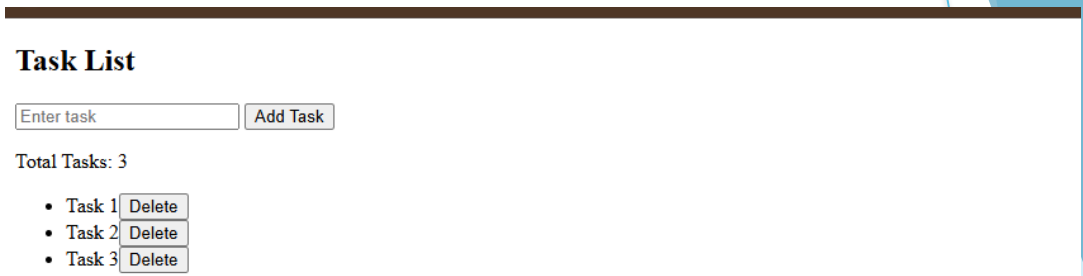
```
document.addEventListener("DOMContentLoaded", function() {
   // your code here
});
```

## Why?

To make sure DOM is fully loaded before JS runs.

# DOM tree

**1-Dynamic To-Do List**
Features:
Add task
Delete task
Mark as completed
Count tasks
Concepts Used:
createElement
addEventListener
classList
remove()




**2-Form Validation System**
Features:
Validate name
Validate email
Validate password
Show error messages dynamically

# Error handling

Error handling, "try...catch"

No matter how great we are at programming, sometimes our scripts have errors. They may occur because of our mistakes, an unexpected user input, an erroneous server response, and for a thousand other reasons.

Usually, a script “dies” (immediately stops) in case of an error, printing it to console.

But there’s a syntax construct try...catch that allows us to “catch” errors so the script can, instead of dying, do something more reasonable.

The “try…catch” syntax

The try...catch construct has two main blocks: try, and then catch:

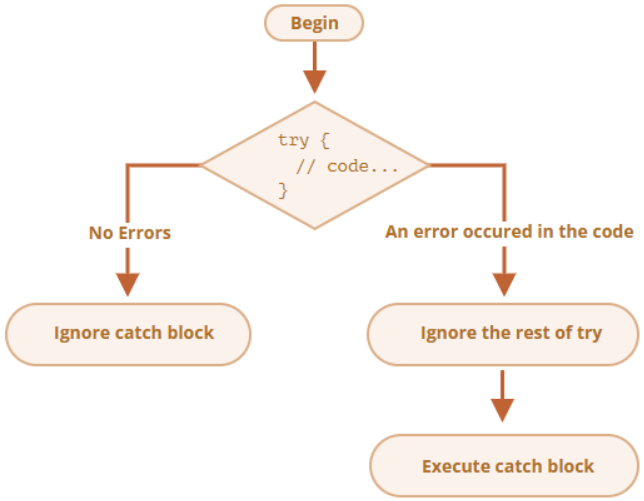
```
try {// code...}
catch (err) {// error handling}
```

# Error handling

Error handling, "try...catch"

It works like this:

1. First, the code in try {...} is executed.
2. If there were no errors, then catch (err) is ignored: the execution reaches the end of try and goes on, skipping catch.
3. If an error occurs, then the try execution is stopped, and control flows to the beginning of catch (err). The err variable (we can use any name for it) will contain an error object with details about what happened.



# Error handling

Error handling, "try...catch"

So, an error inside the try {...} block does not kill the script – we have a chance to handle it in catch.

**Let's look at some examples.**

- An errorless example: shows alert (1) and (2):

```
try {alert('Start of try runs');  // (1) // ...no errors here
alert('End of try runs');   // (2) }
catch (err) {alert('Catch is ignored, because there are no errors'); // (3)}
```

- An example with an error: shows (1) and (3):

```
try {alert('Start of try runs');  // (1)
 lalala; // error, variable is not defined!
alert('End of try (never reached)');  // (2)}
 catch (err) {alert(`Error has occurred!`); // (3) }
```

# Error handling

try…catch…finally

The try...catch construct may have one more code clause: finally.

If it exists, it runs in all cases:

- after try, if there were no errors,
- after catch, if there were errors.

**The extended syntax looks like this:**

try {... try to execute the code ...}

catch (err) {... handle errors ...}

finally {... execute always ...}

**Try running this code:**

```
try {alert( 'try' );
if (confirm('Make an error?')) BAD_CODE();}
catch (err) {alert( 'catch' );} finally {alert( 'finally' );}
```

# Error handling

try…catch…finally

Example:

```
function divide(a, b) {
  try {
    if (b === 0) {
      // Throw an error if the divisor is zero
      throw new Error("Division by zero is not allowed.");
    }
    const result = a / b;
    console.log(`The result is: ${result}`);
    return result;
  } catch (error) {
    // Handle the thrown error
    console.error(`An error occurred: ${error.message}`);
    // You could also log the full error object or notify a service
  } finally {
    // This runs after try or catch, always.
    console.log("Division attempt complete.");
  }
}

divide(10, 2);  // The result is: 5, Division attempt complete.
divide(10, 0);  // An error occurred: Division by zero is not allowed., Division attempt complete.
```

# Task

**Write a function that throws an error if the password is less than 8 characters.**

# What is Scope?

Scope determines where a variable can be accessed in your code.

Types of Scope:

- Global Scope
- Function Scope
- Block Scope (let & const)
- Lexical Scope

# Types of Scope

- **Global Scope**

A variable defined outside any function or block.

```
let name = "Sandy"; // global

function showName() {
    console.log(name); // "Sandy"
}
showName()
```

```
let counter = 0;

function increase() {
    counter++;
}

increase();
increase();
console.log(counter);
```

# Types of Scope

- **Function Scope**

Variables inside a function are only accessible inside it.

```
function test() {
    let message = "Hello";
    console.log(message);
}
test()
console.log(message);
```

```
function gradeStudent(grade) {
    let result;

    if (grade > 90) {
        result = "A";
    }
    return result;
}

console.log(gradeStudent(10));
console.log(result);
```

=> **Make this print not get error**

# Types of Scope

- **Block Scope (let & const)**

Variables inside { } (if, for, while) work only inside that block.

```
for (let i = 1; i <= 3; i++) {
    console.log("Inside:", i);
}
console.log(i);
```

**=> Output?**

# Types of Scope

**Example :**

```
// Global Scope
const globalVar = "I'm global";

function exampleScope() {
  // Function (Local) Scope
  var functionVar = "I'm local to the function";
  console.log(globalVar);

  if (true) {
    // Block Scope (using let)
    let blockVar = "I'm only in this block";
    console.log(functionVar);
    console.log(blockVar);
  }

  console.log(blockVar); // ERROR:
}

exampleScope();
console.log(functionVar); // ERROR:
```

# Types of Scope

- **Lexical Scope**

Inner functions can access variables from outer functions.

**Example:**

```
function outer() {
    let value = 100;

    function inner() {
        console.log(value);
    }

    inner();
}
outer()
```

**=>  Output?**

# Types of Scope

**Example :**

```
function bankAccount() {
    let balance = 500;

    function withdraw(amount) {
        balance -= amount;
        console.log("New balance:", balance);
    }

    return withdraw;
}

const atm = bankAccount();
atm(100);
atm(50);
```

**=> Output?**

# Types of Scope

**Example :**

```
let x = 1;

function levelOne() {
    let x = 2;

    function levelTwo() {
        let x = 3;

        function levelThree() {
            console.log(x);
        }

        return levelThree;
    }

    return levelTwo;
}

const result = levelOne();
const final = result();
final();
```

**=> Output and why?**

# Types of Scope

**Example :**

```
function makeFunctions() {
    let arr = [];

    for (var i = 0; i < 3; i++) {
        arr.push(function () {
            console.log(i);
        });
    }

    return arr;
}

const funcs = makeFunctions();

funcs[0]();
funcs[1]();
funcs[2]();
```

**=> Output and why?**

# What is Closures ?

A closure happens when an inner function remembers variables from its outer function even after the outer function finishes.

**Example :**

```
function createCounter() {
    let count = 0;

    return function () {
        count++;
        console.log(count);
    };
}

const counter = createCounter();
counter(); // 1
counter(); // 2
```

# What is Closures ?

**Example:**

```
function createUser(name) {
    let loggedIn = false;

    return {
        login: function() {
            loggedIn = true;
            console.log(name + " logged in");
        },
        status: function() {
            console.log("Logged in:", loggedIn);
        }
    };
}

const user = createUser("Sandy");
user.status();
user.login();
user.status();
```

=> **Output?**

# Closures Task

Create a function that stores a secret message and returns 2 methods:

- updateMessage()
- readMessage()

# What is Hoisting?

- JavaScript moves variable and function declarations to the top of the scope before execution.

var is hoisted with “undefined”

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

let & const are hoisted but NOT initialized

```
console.log(b); // ❌ Error
let b = 20;
```

# What is Hoisting?

- **JavaScript moves variable and function declarations to the top of the scope before execution.**

Function Declarations are hoisted

```
sayHello(); // works

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

Function Expressions are NOT hoisted

```
greet(); // ❌ Error

const greet = function () {
    console.log("Hi");
};
```

***Thank you***