WEB TECHNOLOGIES

CAP 756

Presenter: Dr. Shuja Mirza

JavaScript Multidimensional Array

- A multidimensional array is an array that contains another array. For example,

```
// multidimensional array
const data = [[1, 2, 3], [1, 3, 4], [4, 5, 6]];
```

- **Create a Multidimensional Array**
- **Example 1**

```
let studentsData = [['Jack', 24], ['Sara', 23], ['Peter', 24]];
```

- **Example 2**

```
let student1 = ['Jack', 24];
let student2 = ['Sara', 23];
let student3 = ['Peter', 24];
// multidimensional array
let studentsData = [student1, student2, student3];
```

- **Access Elements of an Array**
- You can access the elements of a multidimensional array using indices **(0, 1, 2 …)**. For example,

```
let x = [
['Jack', 24],
['Sara', 23],
['Peter', 24]
];

// access the first item
console.log(x[0]); // ["Jack", 24]

// access the first item of the first inner array
console.log(x[0][0]); // Jack

// access the second item of the third inner array
console.log(x[2][1]); // 24
```

	Column 1	Column 2
Row 1	Jack x[0][0]	24 x[0][1]
Row 2	Sara x[1][0]	23 x[1][1]
Row 3	Peter x[2][0]	24 x[2][1]

Add an Element to a Multidimensional Array

- You can use the Array's push() method or an indexing notation to add elements to a multidimensional array.
- **Adding Element to the Outer Array**

```
let studentsData = [['Jack', 24], ['Sara', 23],];
studentsData.push(['Peter', 24]);

console.log(studentsData); //[["Jack", 24], ["Sara", 23], ["Peter", 24]
```

- **Adding Element to the Inner Array**

```
// using index notation
let studentsData = [['Jack', 24], ['Sara', 23],];
studentsData[1][2] = 'hello';

console.log(studentsData); // [["Jack", 24], ["Sara", 23, "hello"]]
```

Remove an Element from a Multidimensional Array

- You can use the Array's pop() method to remove the element from a multidimensional array. For example,

- **Remove Element from Outer Array**

```
// remove the array element from outer array
let studentsData = [['Jack', 24], ['Sara', 23],];
studentsData.pop();

console.log(studentsData); // [["Jack", 24]]
```

- **Remove Element from Inner Array**

```
// remove the element from the inner array
let studentsData = [['Jack', 24], ['Sara', 23]];
studentsData[1].pop();

console.log(studentsData); // [["Jack", 24], ["Sara"]]
```

JavaScript String

- JavaScript string is a primitive data type that is used to work with texts. For example,
- **const name = 'John’;**
- **Create JavaScript Strings**
- In JavaScript, strings are created by surrounding them with quotes. There are three ways you can use quotes.
- **Single quotes: 'Hello'**
- **Double quotes: "Hello"**
- **Backticks: `Hello`**
- For example,
- **//strings example**
- **const name = 'Peter';**
- **const name1 = "Jack";**
- **const result = `The names are ${name} and ${name1}`;**

- You can also write a quote inside another quote. For example,
- **const name = 'My name is "Peter".’;**
- However, the quote should not match the surrounding quotes. For example,
- **const name = 'My name is 'Peter'.'; // error**
- **Access String Characters**
- You can access the characters in a string in two ways.
- One way is to treat strings as an array. For example,
- **const a = 'hello';**
- **console.log(a[1]); // "e“**
- **Another way is to use the method charAt(). For example,**
- **const a = 'hello';**
- **console.log(a.charAt(1)); // "e"**

- **Access String Characters**
- You can access the characters in a string in two ways:
- One way is to treat strings as an array. For example,

```
const a = 'hello';
console.log(a[1]); // "e“
```

- Another way is to use the method charAt(). For example,

```
const a = 'hello';
console.log(a.charAt(1)); // "e“
```

- **JavaScript Strings are immutable**
- In JavaScript, strings are immutable.
- That means the characters of a string cannot be changed. For example,

```
let a = 'hello';
a[0] = 'H';
console.log(a); // "hello“
```

- However, you can assign the variable name to a new string. For example,

```
let a = 'hello';
a = 'Hello';
console.log(a); // "Hello“
```

- **JavaScript is Case-Sensitive**
- JavaScript is case-sensitive. That means in JavaScript, the lowercase and uppercase letters are treated as different values. For example,

```
const a = 'a';
const b = 'A'
console.log(a === b); // false
```

JavaScript ES6

- JavaScript **ES6** (also known as **ECMAScript 2015** or **ECMAScript 6**) is the newer version of JavaScript that was introduced in 2015.
- ECMAScript is the standard that JavaScript programming language uses. ECMAScript provides the specification on how JavaScript programming language should work.

- **JavaScript let**
- JavaScript **let** is used to declare variables. Previously, variables were declared using the **var** keyword.
- The variables declared using **let** are block-scoped.
- This means they are only accessible within a particular block.
- **JavaScript const**
- The **const** statement is used to declare constants in JavaScript. For example,
- Once declared, you cannot change the value of a **const** variable.

JavaScript Arrow Function

- **Arrow function** is one of the features introduced in the ES6 version of JavaScript.
- It allows you to create functions in a cleaner way compared to regular functions. For example,

```
// function expression
let x = function(x, y) {
    return x * y;
}
```

- can be written using an arrow function as:

```
// using arrow functions
let x = (x, y) => x * y;
```

- **Arrow Function Syntax**
- The syntax of the arrow function is:

```
let myFunction = (arg1, arg2, ...argN) => {
    statement(s)
}
```

- If the body has single statement or expression, you can write arrow function as:

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

- **Example 1: Arrow Function with No Argument**
- If a function doesn't take any argument, then you should use empty parentheses. For example,

```
let greet = () => console.log('Hello');
greet(); // Hello
```

- **Example 2: Arrow Function with One Argument**
- If a function has only one argument, you can omit the parentheses. For example,

```
let greet = x => console.log(x);
greet('Hello'); // Hello
```

- **Example 3: Arrow Function as an Expression**
- You can also dynamically create a function and use it as an expression. For example,

```
let age = 5;

let welcome = (age < 18) ?
   () => console.log('Baby') :
   () => console.log('Adult');

welcome(); // Baby
```

JavaScript Default Parameters

- The concept of default parameters is a new feature introduced in the **ES6** version of JavaScript. This allows us to give default values to function parameters

```
function sum(x = 3, y = 5) {

    // return sum
    return x + y;
}

console.log(sum(5, 15));  // 20
console.log(sum(7));      // 12
console.log(sum());       // 8
```

Case 1: Both Argument are Passed

```
sum(5, 15);

function sum(x = 3, y = 5) {
    return x + y;
}
```

Case 2: One Argument is Passed

```
sum(7);

function sum(x = 3, y = 5) {
    return x + y;
}
```

Case 3: No Argument is Passed

```
sum();

function sum(x = 3, y = 5) {
    return x + y;
}
```

- **Using Expressions as Default Values**
- It is also possible to provide expressions as default values.
- **Example 1: Passing Parameter as Default Values**

```
function sum(x = 1, y = x,  z = x + y) {
    console.log( x + y + z );
}

sum(); // 4
```

If you reference the parameter that has not been initialized yet, you will get an error. For example,

```
function sum( x = y, y = 1 ) {
    console.log( x + y);
}

sum();
```

```
ReferenceError: Cannot access 'y' before initialization
```

- **Example 2: Passing Function Value as Default Value**

```
// using a function in default value expression

const sum = () => 15;

const calculate = function( x, y = x * sum() ) {
    return x + y;
}

const result = calculate(10);
console.log(result);              // 160
```

Creating JavaScript Class

- JavaScript class is similar to the Javascript constructor function, and it is merely a syntactic sugar.
- ES6 introduced a new syntax for declaring a class.
- Instead of using the function keyword, you use the class keyword for creating JS classes. For example,

```
class Person {
    constructor(name) {
        this.name = name;
    }
    getName() {
        return this.name;
    }
}
```

In the Person class, the **constructor()** is where you can initialize the properties of an instance. JavaScript automatically calls the **constructor()** method when you instantiate an object of the class.

✓ A JavaScript class is **not** an object.
✓ It is a **template** for JavaScript objects.

- The following creates a new Person object, which will automatically call the constructor() of the Person class:
- **let john = new Person("John Doe");**
- The **getName()** is called a method of the Person class. Like a constructor function, you can call the methods of a class using the following syntax:
- **objectName.methodName(args)**
- For example:
- let name = john.getName();
- console.log(name); // "John Doe“
- To verify the fact that classes are special functions, you can use the typeof operator of to check the type of the Person class.
- **console.log(typeof Person);** // **function**

The Constructor Method is a special method:

- It has to have the exact name "**constructor**"
- It is executed automatically when a new object is created
- It is used to initialize object properties
- If you do not define a constructor method, JavaScript will add an empty constructor method.
- **Class Methods**
- Class methods are created with the same syntax as object methods.
- Use the keyword **class** to create a class.
- Always add a **constructor()** method.
- Then add any number of methods.

- Syntax

```
class ClassName {
  constructor() { ... }
  method_1() { ... }
  method_2() { ... }
  method_3() { ... }
}
```

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Class Method</h2>

<p>How to define and use a Class method.</p>

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

<script>
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;

  }
  age() {
    let date = new Date();
    return date.getFullYear() - this.year;
  }
}

let myCar = new Car("Ford", 2014);
document.getElementById("demo").innerHTML =
"My car is " + myCar.age() + " years old.";
</script>

</body>
</html>
```

JavaScript Modules

- As our program grows bigger, it may contain many lines of code. Instead of putting everything in a single file, you can use modules to separate codes in separate files as per their functionality.
- This makes our code organized and easier to maintain. JavaScript modules allow you to break up your code into separate files.
- JavaScript modules rely on the **import** and **export** statements.
- **Named Exports**: You can create named exports two ways. In-line individually, or all at once at the bottom.
- In-line individually:

```
export const name = "Jesse";
export const age = 40;
```

Person.js

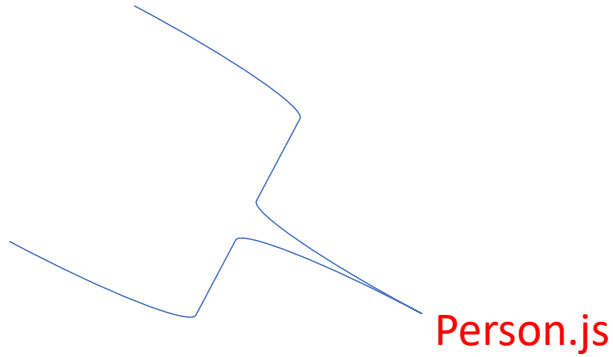
- All at once at the bottom:

```
const name = "Jesse";
const age = 40;
export {name, age};
```

Person.js

- Default Exports
- In this type You can only have one default export in a file.
- ```
  const message = () => {
  const name = "Jesse";
  const age = 40;
  return name + ' is ' + age + 'years old.';
  };

  export default message;
  ```


  ```

Import

- You can import modules into a file in two ways, based on if they are **named exports or default exports.**
- To import a module, we need to use the **import** keyword. The values which are exported from the module can be imported by using the **import** keyword.
- We can import the exported variables, functions, and classes in another module. To import a module, we simply have to specify their path.
- Named exports are constructed using curly braces. Default exports are not.
- Import from named exports
- `import { name, age } from "./person.js";`
- Import from default exports
- Import a default export from the some file named message.js:
- Lets see an example

- Import from named exports

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Modules</h1>

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

<script type="module">
import { name, age } from "./person.js";

let text = "My name is " + name + ", I am " + age + ".";

document.getElementById("demo").innerHTML = text
</script>

</body>
</html>
```

Import from default exports

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Modules</h1>

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

<script type="module">
import message from "./message.js";

document.getElementById("demo").innerHTML = message();

</script>

</body>
</html>
```

- Suppose, a file named **greet.js** contains the following code:

```
// exporting a function
export function greetPerson(name) {
    return `Hello ${name}`;
}
```

- Now, to use the code of **greet.js** in another file, you can use the following code:

```
// importing greetPerson from greet.js file
import { greetPerson } from './greet.js';

// using greetPerson() defined in greet.js
let displayName = greetPerson('Jack');

console.log(displayName); // Hello Jack
```

- **Export Multiple Objects**
- It is also possible to export multiple objects from a module. For example, In the file **module.js**

```
// exporting the variable
export const name = 'JavaScript Program';

// exporting the function
export function sum(x, y) {
    return x + y;
}
```

- In main file,

```
import { name, sum } from './module.js';

console.log(name);
let add = sum(4, 9);
console.log(add); // 13
```

This imports both the name variable and the sum() function from the module.js file.

ES6 Promises

- A Promise is a JavaScript object that links "Producing Code" and "Consuming Code".
- "Producing Code" can take some time and "Consuming Code" must wait for the result.
- Promise is the easiest way to work with asynchronous programming in JavaScript.
- Asynchronous programming includes the running of processes individually from the main thread and notifies the main thread when it gets complete.
- Prior to the Promises, Callbacks were used to perform asynchronous programming.

• How Does Promise work?

- The Promise represents the completion of an asynchronous operation. It returns a single value based on the operation being rejected or resolved.

- A promise may have one of three states.

1. Pending
2. Fulfilled
3. Rejected

- A promise starts in a **pending** state. That means the process is not complete. If the **operation is successful**, the process ends in a **fulfilled** state. And, if an **error occurs**, the process ends in a **rejected state**.
- **Pending -** It is the initial state of each Promise. It represents that the result has not been computed yet.
- **Fulfilled -** It means that the operation has completed.
- **Rejected -** It represents a failure that occurs during computation.
- Once a Promise is fulfilled or rejected, it will be immutable. The **Promise()** constructor takes two arguments that are **rejected** function and a **resolve** function.
- Based on the asynchronous operation, it returns either the first argument or second argument.

- For example, when you request data from the server by using a promise, it will be in a pending state.
- When the data arrives successfully, it will be in a fulfilled state.
- If an error occurs, then it will be in a rejected state.
- **Create a Promise**
- To create a promise object, we use the **Promise()** constructor.

```
let promise = new Promise(function(resolve, reject){
      //do something
});
```

- **Example 1: Program with a Promise**

```
const count = true;

let countValue = new Promise(function (resolve, reject) {
    if (count) {
        resolve("There is a count value.");
    } else {
        reject("There is no count value");
    }
});

console.log(countValue);
```

Output

```
Promise {<resolved>: "There is a count value."}
```

JavaScript Promise Chaining

- Promises are useful when you have to handle more than one asynchronous task, one after another. For that, we use promise chaining.
- You can perform an operation after a promise is resolved using methods **then()**, **catch()** and **finally().**
- **JavaScript then() method**
- The **then()** method is used with the callback when the promise is successfully fulfilled or resolved**.**
- **The syntax of then() method is:**
- **promiseObject.then(onFulfilled, onRejected);**
- **Example : Chaining the Promise with then() follows…..**

```
// returns a promise

let countValue = new Promise(function (resolve, reject) {
  resolve("Promise resolved");
});

// executes when promise is resolved successfully

countValue
  .then(function successValue(result) {
    console.log(result);
  })

  .then(function successValue1() {
    console.log("You can call multiple functions this way.");
  });
```

```
Promise resolved
You can call multiple functions this way.
```

- **JavaScript catch() method**
- The catch() method is used with the callback when the promise is rejected or if an error occurs. For example,

```
// returns a promise
let countValue = new Promise(function (resolve, reject) {
   reject('Promise rejected');
});

// executes when promise is resolved successfully
countValue.then(
    function successValue(result) {
        console.log(result);
    },
 )

// executes if there is an error
.catch(
    function errorValue(result) {
        console.log(result);
    }
);
```

Output

```
Promise rejected
```

- **JavaScript finally() method**

```
// returns a promise
let countValue = new Promise(function (resolve, reject) {
    // could be resolved or rejected
    resolve('Promise resolved');
});

// add other blocks of code
countValue.finally(
    function greet() {
        console.log('This code is executed.');
    }
);
```

- **Output**

```
This code is executed.
```

CommonJS:

- CommonJS is a popular modularization pattern that's used in Node.js.
- The CommonJS system is centered around a **require()** function that loads other modules and an exports property that lets modules export publicly accessible methods.
- ***How can I use CommonJS?***
- CommonJS wraps each module in a function called ‘require’, and includes an object called ‘module.exports’, which exports code for availability to be required by other modules***.***
- ***Why would I need CommonJS??***
- CommonJS allows for code encapsulation, as modules with no global variables won’t conflict with each other when your application is run

- **What is Imperative Programming?**
- Imagine we have a list of the world's most commonly-used passwords:

```
const passwords = [
    "123456",
    "password",
    "admin",
    "freecodecamp",
    "mypassword123",
];
```

- Imperatively, we would write:

```
// using the passwords constant from above

let longPasswords = [];
for (let i = 0; i < passwords.length; i++)
{
    const password = passwords[i];
    if (password.length >= 9) {
        longPasswords.push(password);
    }
}

console.log(longPasswords);
// logs ["myusername", "mypassword123"];
```

What is Declarative Programming?

- Imperative and declarative programming achieve the same goals.
- They are just different ways of thinking about code.
- They have their benefits and drawbacks and there are times to use both.
- So instead of giving the computer step by step instructions, we declare what it is we want and we assign this to the result of some process.

```
// using the passwords constant from above

const longPasswords = passwords.filter(password => password.length >= 9);

console.log(longPasswords); // logs ["myloginname", "mypassword123"];
```

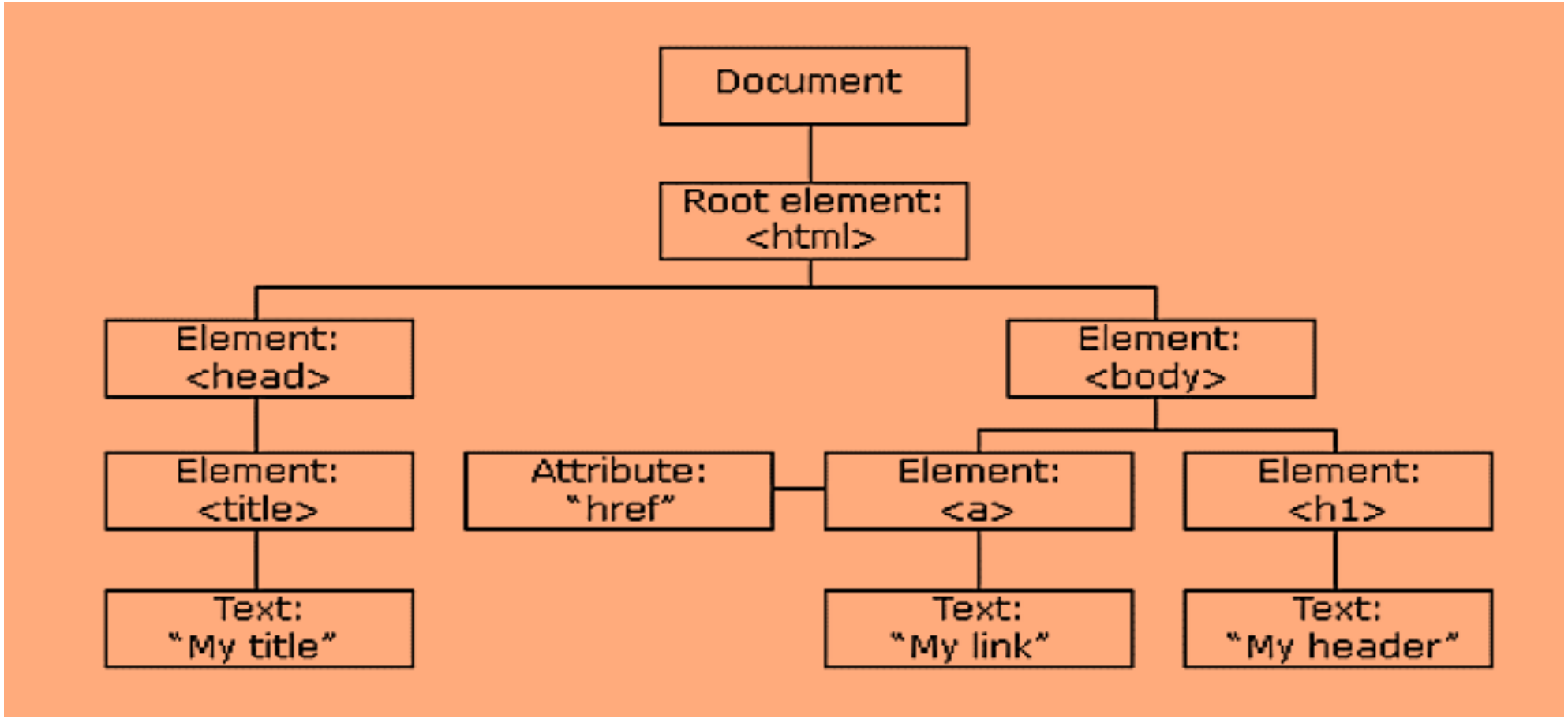
Imperative	Declarative
How to do things.	What to do.
Statements.	Expressions.
Defining variables and changing their values.	Evaluate result based on input.

```
const expression = input => input.toLowerCase();
const expression2 = function(input) {
  return input.toLowerCase();
};

const statement = () => console.log('hello world');
const statement2 = function() {
  console.log('hello world');
};
```

JavaScript DOM

- With the HTML DOM, JavaScript can access and change all the elements of an HTML document.
- The **HTML DOM** model is constructed as a tree of **Objects**:



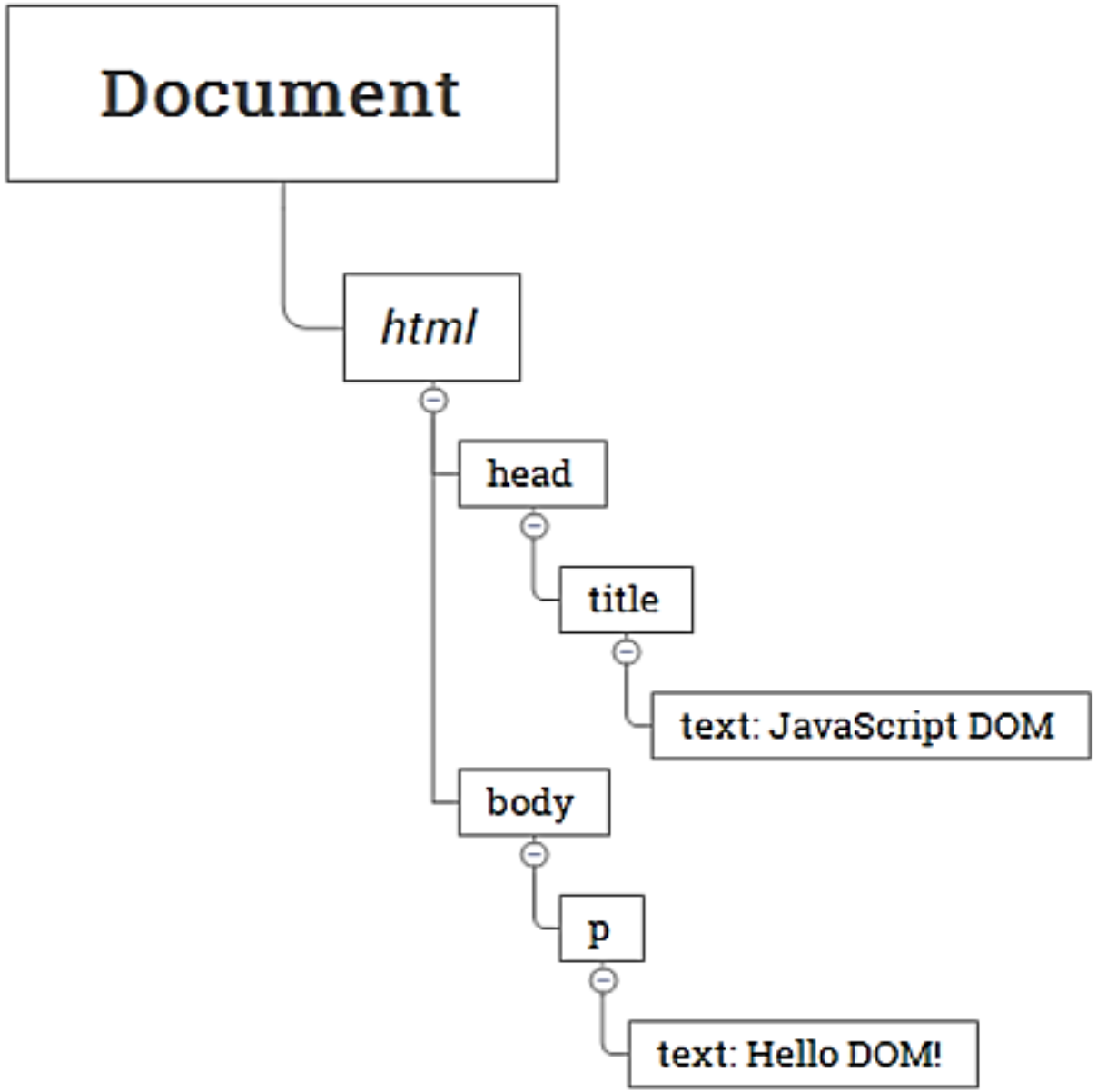
The DOM Programming Interface

- The HTML DOM can be accessed with JavaScript (and with other programming languages).
- Most often, you want to manipulate HTML elements. To manipulate element you have to find the elements first.
- Javascript provides us with various methods to find an element within the document.
- In the DOM, all HTML elements are defined as **objects**.
- A **property** is a value that you can get or set (like changing the content of an HTML element).
- A **method** is an action you can do (like add or deleting an HTML element). Example
- The following example changes the content (the innerHTML) of the
- <p> element with id="demo":

- <!DOCTYPE html>
- <html>
- <body>
- <h2>My First Page</h2>
-
- <script>
- document.getElementById("demo").innerHTML = "Hello World!";
- </script>
- </body>
- </html>

```
<html>
    <head>
        <title>JavaScript DOM</title>
    </head>
    <body>
        <p>Hello DOM!</p>
    </body>
</html>
```

- The tree represents this HTML document:



Node Types

- Each node in the DOM tree is identified by a node type. JavaScript uses integer numbers to determine the node types.

Constant	Value	Description
`Node.ELEMENT_NODE`	1	An `Element` node like `<p>` or `<div>`.
`Node.TEXT_NODE`	3	The actual `Text` inside an `Element` or `Attr`.
`Node.CDATA_SECTION_NODE`	4	A `CDATASection`, such as `<!CDATA[[ … ]]>`.
`Node.PROCESSING_INSTRUCTION_NODE`	7	A `ProcessingInstruction` of an XML document, such as `<?xml-stylesheet … ?>`.
`Node.COMMENT_NODE`	8	A `Comment` node, such as `<!-- … -->`.
`Node.DOCUMENT_NODE`	9	A `Document` node.
`Node.DOCUMENT_TYPE_NODE`	10	A `DocumentType` node, such as `<!DOCTYPE html>`.
`Node.DOCUMENT_FRAGMENT_NODE`	11	A `DocumentFragment` node.

- To get the type of node, you use the **nodeType** property: node.nodeType
- You can compare the **nodeType** property with the above constants to determine the node type. For example:

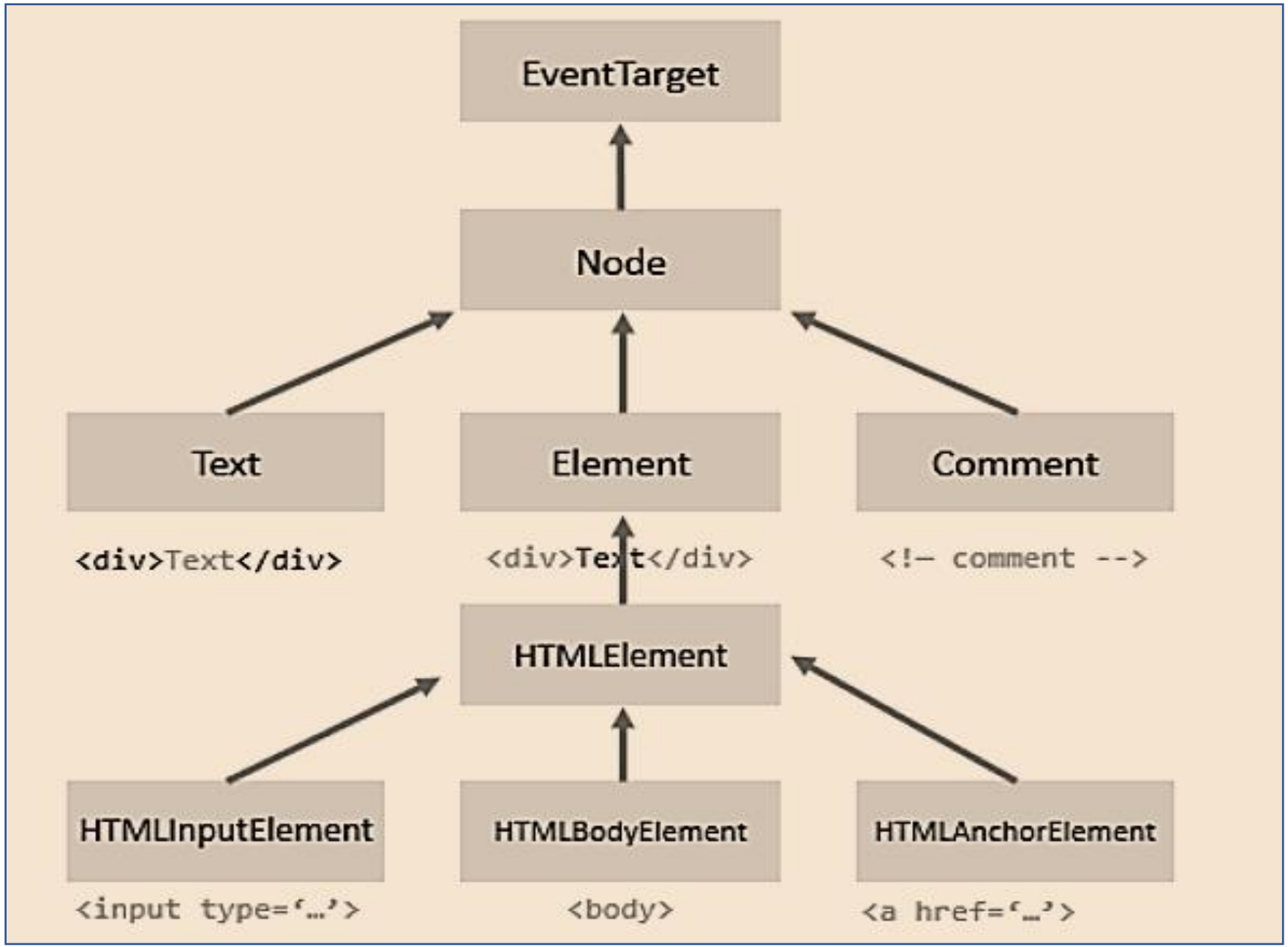
```
if (node.nodeType == Node.ELEMENT_NODE) {
    // node is the element node
}
```

- The **nodeName** and **nodeValue** properties
- A node has two important properties: nodeName and nodeValue that provide specific information about the node.
- The values of these properties depend on the node type.
- For example, if the node type is the element node, the nodeName is always the same as the element's tag name and nodeValue is always null.

```
if (node.nodeType == Node.ELEMENT_NODE) {
    let name = node.nodeName; // tag name like <p>
}
```

- Node and Element
- Sometimes it’s easy to confuse between the Node and the Element.
- A **node** is a generic name of any object in the **DOM** tree.
- It can be any built-in DOM element such as the document. Or it can be any HTML tag specified in the HTML document like or <p>.
- An element is a node with a specific node type Node.ELEMENT_NODE, which is equal to 1.
- In other words, the node is the generic type of element.
- The element is a specific type of the node with the node type Node.ELEMENT_NODE.

- The following picture illustrates the relationship between the Node and Element types:



Node Relationships

- Any node has relationships to other nodes in the DOM tree. The relationships are the same as the ones described in a traditional family tree.
- The following picture illustrates the relationships between nodes: