JS DOM

Part 4

Document Object Model

The **document object** represents the whole html document.

When html document is loaded in the browser, it becomes a document object. It is the **root element** that represents the html document. It has properties and methods. By the help of document object, we can add dynamic content to our web page.

Method	Description
write("string")	writes the given string on the doucment.
writeln("string")	writes the given string on the doucment with newline character at the end.
getElementById()	returns the element having the given id value.
getElementsByName()	returns all the elements having the given name value.
getElementsByTagName()	returns all the elements having the given tag name.
getElementsByClassName()	returns all the elements having the given class name.

The **document.getElementById()** method returns the element of specified id.

```
<script type="text/javascript">
function getcube(){
var number=document.getElementById("number").value;
alert(number*number*number);
}
</script>
<form>
Enter No:<input type="text" id="number" name="number"/><br/>
<input type="button" value="cube" onclick="getcube()"/>
</form>
```

The **document.getElementsByName()** method returns all the element of specified name.

```
<script type="text/javascript">
function totalelements()
{
var allgenders=document.getElementsByName("gender");
alert("Total Genders:"+allgenders.length);
}
</script>
<form>
Male:<input type="radio" name="gender" value="male">
Female:<input type="radio" name="gender" value="female">

<input type="button" onclick="totalelements()" value="Total Genders">
</form>
```

The **document.getElementsByTagName()** method returns all the element of specified tag name.

```
<script type="text/javascript">
function countpara(){
var totalpara=document.getElementsByTagName("p");
alert("total p tags are: "+totalpara.length);


}
</script>
<p>This is a pragraph</p>
<p>Here we are going to count total number of paragraphs by getElementByT
agName() method.</p>
<p>Let's see the simple example</p>
<button onclick="countpara()">count paragraph</button>
```

The **innerHTML** property can be used to write the dynamic html on the html document.
It is used mostly in the web pages to generate the dynamic html such as registration form, comment form, links etc.

```
<script type="text/javascript" >
function showcommentform() {
var data="Name:<input type='text' name='name'><br>Comment:<br><textarea rows='5'
 cols='80'></textarea>
<br><input type='submit' value='Post Comment'>";
document.getElementById('mylocation').innerHTML=data;
}
</script>
<form name="myForm">
<input type="button" value="comment" onclick="showcommentform()">
<div id="mylocation"></div>
</form>
```

The **innerText** property can be used to write the dynamic text on the html document. Here, text will not be interpreted as html text but a normal text.
It is used mostly in the web pages to generate the dynamic content such as writing the validation message, password strength etc.

```
<script type="text/javascript" >
function validate() {
var msg;
if(document.myForm.userPass.value.length>5){
msg="good";
}
else{
msg="poor";
}
document.getElementById('mylocation').innerText=msg;
 }

</script>
<form name="myForm">
<input type="password" value="" name="userPass" onkeyup="validate()">
Strength:<span id="mylocation">no strength</span>
</form>
```

JavaScript Form Validation

JavaScript HTML DOM - Changing CSS

The HTML DOM allows JavaScript to change the style of HTML elements.

```
<html>
<body>

<p id="p2">Hello World!</p>

<script>
document.getElementById("p2").style.color = "blue";
</script>

<p>The paragraph above was changed by a script.</p>

</body>
</html>
```

Using Events

The HTML DOM allows you to execute code when an event occurs.

Events are generated by the browser when "things happen" to HTML elements:

- An element is clicked on
- The page has loaded
- Input fields are changed

```
<body>

<h1 id="id1">My Heading 1</h1>

<button type="button"
onclick="document.getElementById('id1').style.color = 'red'">
Click Me!</button>

</body>
</html>
```

JavaScript HTML DOM Events

Reacting to Events

A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element.

To execute code when a user clicks on an element, add JavaScript code to an HTML event attribute:

- Examples of HTML events:
- When a user clicks the mouse
- When a web page has loaded
- When an image has been loaded
- When the mouse moves over an element
- When an input field is changed
- When an HTML form is submitted
- When a user strokes a key

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

<h1 onclick="changeText(this)">Click on this
text!</h1>

<script>
function changeText(id) {
  id.innerHTML = "Ooops!";
}
</script>

</body>
</html>
```

The onload and onunload Events

The `onload` and `onunload` events are triggered when the user enters or leaves the page.

The `onload` event can be used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.

The `onload` and `onunload` events can be used to deal with cookies.

The onchange Event

The `onchange` event is often used in combination with validation of input fields.

Below is an example of how to use the onchange. The `upperCase()` function will be called when a user changes the content of an input field.

```
<input type="text" id="fname" onchange="upperCase()">
```

```
<!DOCTYPE html>
<html>
<body onload="checkCookies()">

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

<script>
function checkCookies() {
  var text = "";
  if (navigator.cookieEnabled == true) {
    text = "Cookies are enabled.";
  } else {
    text = "Cookies are not enabled.";
  }
  document.getElementById("demo").innerHTML = text;
}
</script>

</body>
</html>
```

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

<div onmouseover="mOver(this)" onmouseout="mOut(this)"
style="background-color:#D94A38;width:120px;height:20px;padding:40px;">
Mouse Over Me</div>

<script>
function mOver(obj) {
  obj.innerHTML = "Thank You"
}

function mOut(obj) {
  obj.innerHTML = "Mouse Over Me"
}
</script>

</body>
</html>
```

JavaScript HTML DOM Animation

```
<!DOCTYPE html>
<html>
<style>
#container {
  width: 400px;
  height: 400px;
  position: relative;
  background: yellow;
}
#animate {
  width: 50px;
  height: 50px;
  position: absolute;
  background-color: red;
}
</style>
```

```
<body>
<p><button onclick="myMove()">Click Me</button></p>

<div id ="container">
  <div id ="animate"></div></div>
<script>
var id = null;
function myMove() {
  var elem = document.getElementById("animate");
  var pos = 0;
  clearInterval(id);
  id = setInterval(frame, 5);
  function frame() {
    if (pos == 350) {
      clearInterval(id);
    } else {
      pos++;
      elem.style.top = pos + "px";
      elem.style.left = pos + "px";
    }  } }
</script>
```

JavaScript timer

In JavaScript, a timer is created to execute a task or any function at a particular time. Basically, the timer is used to delay the execution of the program or to execute the JavaScript code in a regular time interval. So, the code does not complete it's execution at the same time when an event triggers or page loads.

The best example of the timer is advertisement banners on websites, which change after every 2-3 seconds. We set a time interval to change them.

JavaScript offers two timer functions **setInterval()** and **setTimeout()**, which helps to delay in execution of code and also allows to perform one or more operations repeatedly.

setTimeout()

The setTimeout() function helps the users to delay the execution of code. The setTimeout() method accepts two parameters in which one is a user-defined function, and another is the time parameter to delay the execution. The time parameter holds the time in milliseconds (1 second = 1000 milliseconds), which is optional to pass.

```
setTimeout(function, milliseconds)
```

```
<html>
<body>
<script>
function delayFunction() {
    document.write('<h3> Welcome to JS Timer  <h3>');
}
</script>
<h4> Example of delay the execution of function <h4>
<button onclick = "setTimeout(delayFunction, 3000)"> Click Here </button>

</body>
</html>
```

setInterval()

The setInterval method is a bit similar to the setTimeout() function. It executes the specified function repeatedly after a time interval. Or we can simply say that a function is executed repeatedly after a specific amount of time provided by the user in this function.

```
setInterval(function, milliseconds)
```

```
<html>
<body>
<script>
function waitAndshow() {
var systemdate = new Date();
   //display the updated time after every 4 seconds
       document.getElementById("clock").innerHTML = systemdate.toLocaleTimeString();
}
      setInterval(waitAndshow, 4000);
</script>

<h3> Updated time will show in every 4 seconds </h3>
<h3> The current time on your computer is: <br>
<span id="clock"></span> </h3>

</body>
</html>
```

JavaScript callback

A callback function can be defined as a function passed into another function as a parameter. Don't relate the callback with the keyword, as the callback is just a name of an argument that is passed to a function.

"I will call back later!"

A callback is a function passed as an argument to another function

This technique allows a function to call another function

A callback function can run after another function has finished

```
<body>
<h1> Call back illustrator</h1>
<h3> The getData() function is called </h3>
<script>
function getData(x, y, callback){
document.write(" The multiplication of the numbers " + x + " and " + y + " is: " + (x*y) + "<br><br>" );
callback();
}
function showData(){
document.write(' This is the showData() method execute after the completion of getData() method.');
}
getData(20, 30, showData);
</script>
</body>
```

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

Asynchronous JavaScript

"I will finish later!"

Functions running in parallel with other functions are called asynchronous

A good example is JavaScript setTimeout()

There are two main types of asynchronous code style you'll come across in JavaScript code, old-style callbacks and newer promise-style code.

Async callbacks

Async callbacks are functions that are specified as arguments when calling a function which will start executing code in the background. When the background code finishes running, it calls the callback function to let you know the work is done, or to let you know that something of interest has happened.

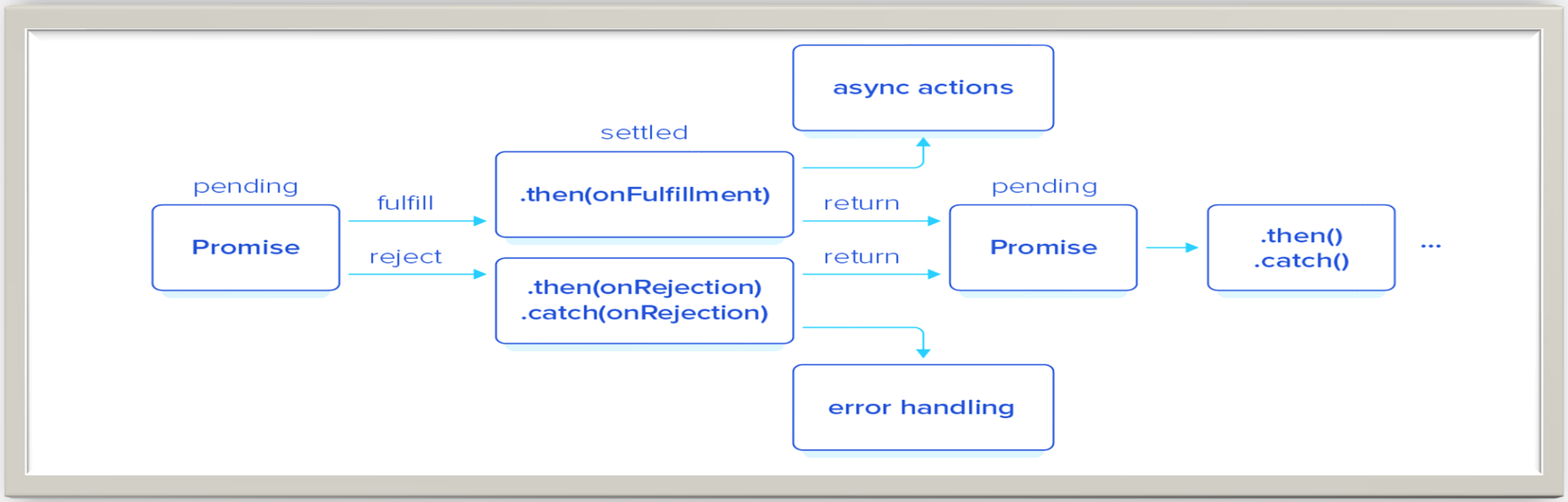
```
<body>
<h2>JavaScript SetTimeout()</h2>

<p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>

<h1 id="demo"></h1>
<script>
setTimeout(function() { myFunction(“Friends for ever !!!"); }, 3000);

function myFunction(value) {  document.getElementById("demo").innerHTML = value; }
</script>

</body>
```

```
<body>

<h2>JavaScript setInterval()</h2>

<p>Using setInterval() to display the time every second (1000 milliseconds).</p>
<h1 id="demo"></h1>

<script>
setInterval(myFunction, 1000);

function myFunction() {
  let d = new Date();
  document.getElementById("demo").innerHTML=
  d.getHours() + ":" + d.getMinutes() + ":" + d.getSeconds();
}
</script>

</body>
```

JavaScript Promises

Promises are the new style of async code that you'll see used in modern Web APIs.

This method did not remove the use of callbacks, but it made the chaining of functions straightforward and simplified the code, making it much easier to read.



```
let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Promise Object Properties

A JavaScript Promise object can be:

- Pending
- Fulfilled
- Rejected

The Promise object supports two properties: **state** and **result**.

While a Promise object is "pending" (working), the result is undefined.

When a Promise object is "fulfilled", the result is a value.

When a Promise object is "rejected", the result is an error object.

You must use a Promise method to handle promises.

```
function myDisplayer(some) {
 console.log(some);
}

let myPromise = new Promise(function(myResolve, myReject) {
  let x = 7;

// some code (try to change x to 5)

  if (x == 0) {
    myResolve("OK");
  } else {
    myReject("Error");
  }
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
```

JavaScript Async

The keyword async before a function makes the function return a promise:

```
async function f() { return 1; }
f().then(alert); // 1
```

So, async ensures that the function returns a promise, and wraps non-promises in it.
There's another keyword, await,
that works only inside async functions

```
function mf(s)
{
    console.log(s)
}
async function f()
{

}
f().then(
    function(value){mf("k")},
    function(error){mf("not")})
```

Await Syntax

The keyword `await` before a function makes the function wait for a promise:

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

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

```
async function f()
 {
let promise = new Promise((resolve, reject) => {
setTimeout(() => resolve("done!"), 1000) });
 let result = await promise;
// wait until the promise resolves (*)
alert(result);
 // "done!"
}
f();
```

The function execution “pauses” at the line `(*)` and resumes when the promise settles, with `result` becoming its result. So the code above shows “done!” in one second.