
Unit 2: HTML5, JQuery And Ajax

The **Document Object Model**, or **DOM** for short, is a platform and language independent model to represent the HTML or XML documents. It defines the logical structure of the documents and the way in which they can be accessed and manipulated by an application program.

In the DOM, all parts of the document, such as elements, attributes, text, etc. are organized in a hierarchical tree-like structure; similar to a family tree in real life that consists of parents and children.

In DOM terminology these individual parts of the document are known as *nodes*. The DOM represents the document as nodes and objects. It is a way programming languages can connect to the page.

The Document Object Model that represents HTML document is referred to as HTML DOM. Similarly, the DOM that represents the XML document is referred to as XML DOM.

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.

A Web page is a document. This document can be either displayed in the browser window or as the HTML source. But it is the same document in both cases.

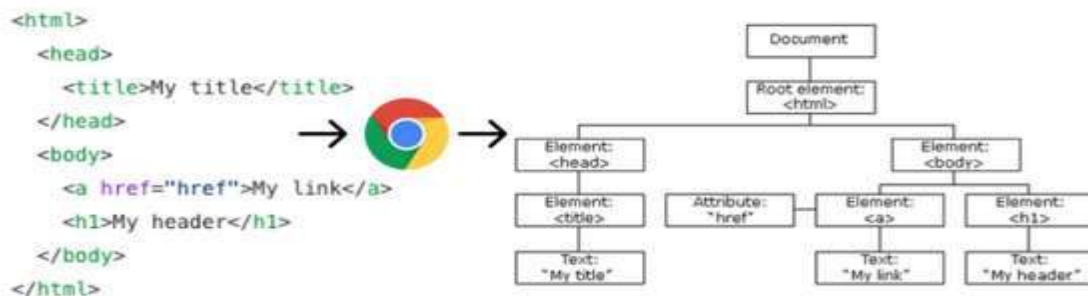
The Document Object Model (DOM) represents that same document so it can be manipulated. The DOM is an object-oriented representation of the web page, which can be modified with a scripting language such as JavaScript.

The DOM is *not*:

- part of the JavaScript language

The DOM is:

- constructed from the browser
- is globally accessible by JavaScript code using the document object



<html>

<head>

<title>JavaScript DOM</title>

</head>

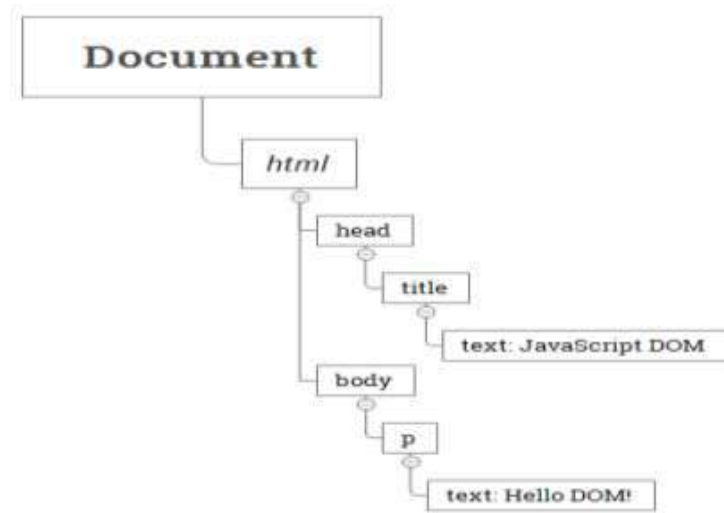
<body>

<p>Hello DOM!</p>

</body>

</html>

The following tree represents the above HTML document:



In this DOM tree, the document is the root node. The root node has one child which is the `<html>` element. The `<html>` element is called the *document element*.

Each document can have only one document element. In an HTML document, the document element is the `<html>` element. Each markup can be represented by a node in the tree.

Whenever an *HTML document* is loaded into a *web browser*, it becomes a document object. The Document Object Model (DOM) represents that same document so it can be manipulated. The DOM is an object-oriented representation of the web page, which can be modified with a scripting language such as JavaScript.

DOM is an object-oriented representation of the web page, which helps us to dynamically modify the web page indirectly, through the DOM, with the help of scripting languages like JavaScript.

In the HTML DOM, everything is referred to as a node. This means, HTML DOM is composed of units called nodes.

For example:

1. The document (web-page) is a document node.
2. All HTML elements (*<head>*, *<body>*) are element nodes.
3. All HTML attributes (*<input type="text" />*) are attribute nodes.
4. Text inside the HTML elements are text nodes.
5. Even, the comments are comment nodes.

In the HTML DOM, HTML elements are referred to as Element Objects. A collection of HTML elements is referred to as a NodeList object (list of nodes).

Every HTML element in a Web page is a scriptable object in the object model, with its own set of properties, methods, and events. To enable access to these objects, Internet Explorer creates a top-level document object for each HTML document it displays. When you use the .Object property on a Web page object in your test or component, you actually get a reference to this DOM object. This document object represents the entire page. From this document object, you can access the rest of the object hierarchy by using properties and collections.

- Methods: Methods can be defined as the actions that you'll apply to your HTML elements.
- Properties: Properties can be defined as the values that you will change, add or remove in some way.

The way a document content is accessed and modified is called the Document Object Model, or DOM. The Objects are organized in a hierarchy. This hierarchical structure applies to the organization of objects in a Web document.

- Window object – Top of the hierarchy. It is the outmost element of the object hierarchy.
- Document object – Each HTML document that gets loaded into a window becomes a document object. The document contains the contents of the page.
- Form object – Everything enclosed in the <form>...</form> tags sets the form object.
- Form control elements – The form object contains all the elements defined for that object such as text fields, buttons, radio buttons, and checkboxes.

The Document Fragment interface is a lightweight version of the Document that stores a piece of document structure like a standard document. However, a Document Fragment is not part of the active DOM tree.

If you make changes to the document fragment, it doesn't affect the document or incurs any performance.

Typically, you use the Document Fragment to compose DOM nodes and append or insert it to the active DOM tree using `appendChild()` or `insertBefore()` method.

To create a new document fragment, you use the Document Fragment constructor like this:

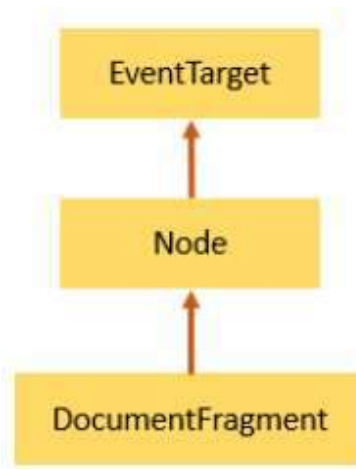
`let fragment = new Document Fragment();`

Or you can use the `createDocumentFragment()` method of the Document object:

`let fragment = document.createDocumentFragment();`

This Document Fragment inherits the methods of its parent, Node, and also implements those of the ParentNode interface such as `querySelector()` and

querySelectorAll(). Use the Document Fragment to compose DOM nodes before updating them to the active DOM tree to get better performance.



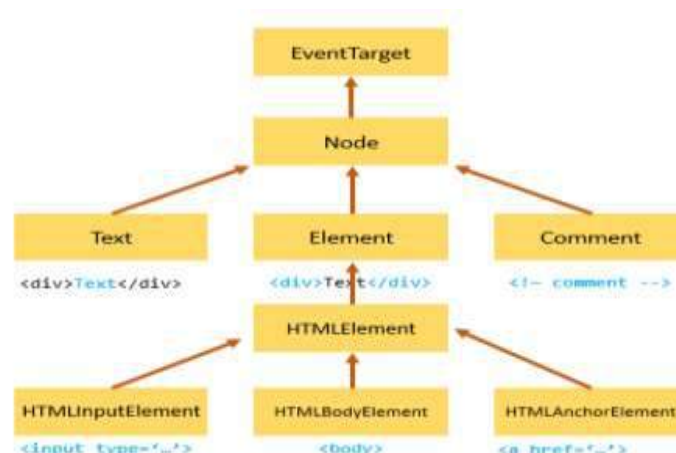
Node and Element

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 `<div>` 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 generic type of the 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:



The `getElementById()` and `querySelector()` returns an object with the `Element` type while `getElementsByName()` or `querySelectorAll()` returns `NodeList` which is a collection of nodes.

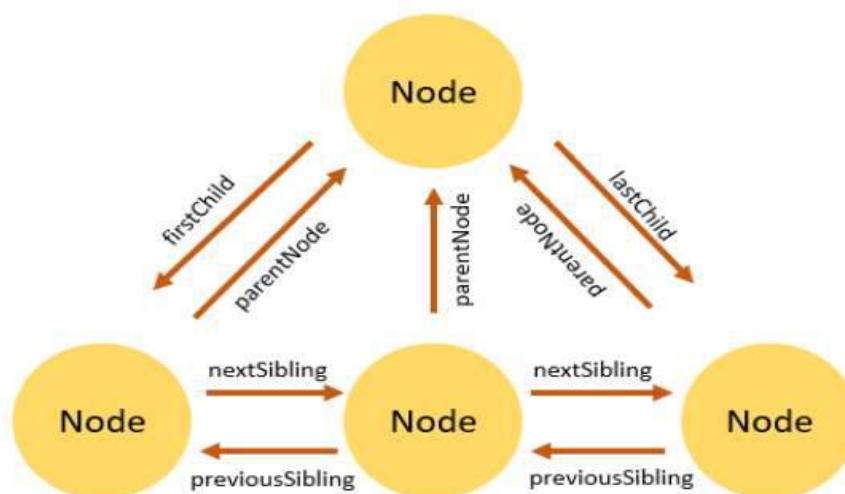
Node Relationships

Any node has relationships to other nodes in the DOM tree. The relationships are the same as the one described in a traditional family tree.

For example, `<body>` is a child node of the `<html>` node, and `<html>` is the parent of the `<body>` node.

The `<body>` node is the sibling of the `<head>` node because they share the same immediate parent, which is the `<html>` element.

The following picture illustrates the relationships between nodes:



Selecting DOM Elements in JavaScript

JavaScript is most commonly used to get or modify the content or value of the HTML elements on the page, as well as to apply some effects like show, hide, animations etc. But, before you can perform any action you need to find or select the target HTML element.

Selecting Elements by ID

You can select an element based on its unique ID with the `getElementById()` method. This is the easiest way to find an HTML element in the DOM tree. The `getElementById()` method will return the element as an object if the matching element was found, or null if no matching element was found in the document. Any HTML element can have an id attribute. The value of this attribute must be unique within a page i.e. no two elements in the same page can have the same ID.

Syntax :

```
var element = document.getElementById(id);
```

Parameters :

- **Id** : The ID of the element to locate. The ID is case-sensitive string which is unique within the document; only one element may have any given ID.
- **Return value** : An Element object describing the DOM element object matching the specified ID, or null if no matching element was found in the document.

NOTE :

- The id is case-sensitive. For example, the 'message' and 'Message' are totally different ids.
- The id also unique in the document. If a document has more than one element with the same id, the `getElementById()` method returns only the first one it encounters.

CODE:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>JS Element by ID</title>
</head>
<body>
  <p id="first">This is a paragraph of text.</p>
  <p>This is another paragraph of text.</p>

  <script>
    // Selecting element with id first
    var match = document.getElementById("first");
    // Highlighting element's background
    match.style.background = "yellow";
  </script>
</body>
</html>
```

Results:

This is a paragraph of text.

This is another paragraph of text.

innerHTML:

The Element property `innerHTML` gets or sets the HTML or XML markup contained within the element. `innerHTML` is a way of accessing the content inside of an XHTML element. It is used with Ajax requests so that choosing an element on the web page, then write the Ajax-loaded content straight into that element via `innerHTML`. `innerHTML` element can also be read directly and appended to.

Syntax :

```
const content = element.innerHTML;
```

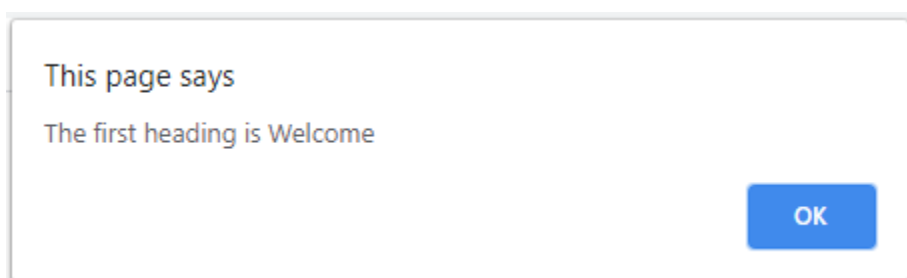
```
element.innerHTML = htmlString;
```

A DOM String containing the HTML serialization of the element's descendants. Setting the value of innerHTML removes all of the element's descendants and replaces them with nodes constructed by parsing the HTML given in the string htmlString.

CODE:

```
<html>
<head>
  <title>InnerHTML</title>
</head>
<body>
  <h1 id="one">Welcome</h1>
  <p>This is the welcome message.</p>
  <h2>Technology</h2>
  <p>This is the technology section.</p>
  <script type="text/javascript">
    var text = document.getElementById("one").innerHTML;
    alert("The first heading is " + text);
  </script>
</body>
</html>
```

Results:



Onclick the Ok button :

Welcome

This is the welcome message.

Technology

This is the technology section.

Selecting Elements by Name

Elements on an HTML document can have a name attribute. Unlike the id attribute, multiple element can share the same value of the name attribute.

To get all elements with a specified name, you use the `getElementsByName()` method of the document object:

Syntax:

```
let elements = document.getElementsByName(name);
```

The `getElementsByName()` accepts a name which is the value of the name attribute of elements and returns a live `NodeList` of elements.

The return collection of elements is live. It means that the elements are automatically updated when elements with the same name are added and/or removed from the document.

CODE:

```

<!DOCTYPE html>-
<html>-

<head>-
....<title>getElementsByName()</title>-
....<style>-
.....body {-
.....    text-align: center;-
.....}-
.....
.....h1 {-
.....    color: green;-
.....}-
....</style>-
....<script>-
.....// creating function to display
.....// elements at particular name
.....function display() {-
.....
.....    // This line will print entered result
.....    alert(document.getElementsByName("ga")[0].value);
.....}
....</script>-
</head>-

```

```

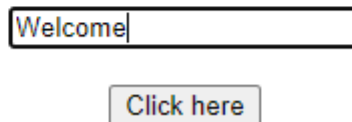
<body>-
....<h1>display screen</h1>-
....<h2>Example for getElementsByName() Method</h2>-
....<!-- This will create an input tag-->
....<input type="text" name="ga" />-
....<br>-
....<br>-
....<!-- function will be called when we
....click on this button-->
....<input type="button" onclick="display()"
....|-----|-----|-----|-----value="Click here" />-
....<p></p>-
</body>-
</html>-

```

Results:

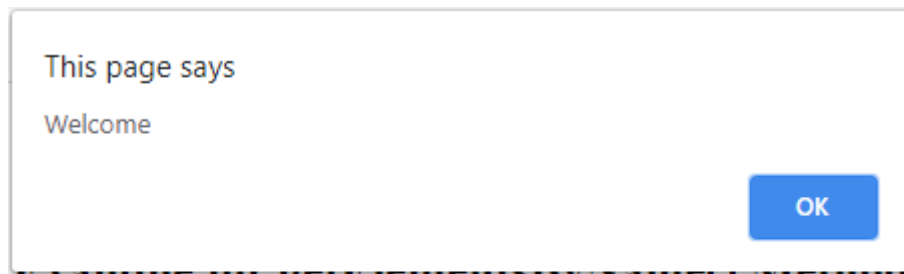
display screen

Example for getElementByName() Method



A screenshot of a web form. It features a text input field with the text 'Welcome' inside. Below the input field is a button labeled 'Click here'.

Once you click on CLICK HERE button the alert box pops up:



Example for getElementByName() Method



A screenshot of a web form, identical to the one above. It has a text input field with 'Welcome' and a 'Click here' button below it.

Selecting Elements by Class Name

Similarly, you can use the `getElementsByClassName()` method to select all the elements having specific class names. This method returns an array-like object of all child elements which have all of the given class names.

Syntax:

```
var ele=document.getELeMentsByClassName('name');
```

CODE:

```
<!DOCTYPE html>
<html>
<head>
....<meta charset="utf-8">
....<title>JS Select Element by CLASS</title>
</head>
<body>
....<p class="first">This is a paragraph of text.</p>
....<p>This is another paragraph of text.</p>

....<script>
....// Selecting element with id first.
....var match = document.getElementsByClassName("first");
.....
...</script>
</body>
</html>
```

Results:

This is a paragraph of text.

This is another paragraph of text.

Selecting Elements by Tag Name

You can also select HTML elements by tag name using the `getElementsByTagName()` method. This method also returns an array-like object of all child elements with the given tag name.

Syntax :

```
var x = document.getElementsByTagName("p");
```

CODE:

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

<h2>JS Elements by Tag Name</h2>

<p>A bad workman always blames his tools!</p>
<p>Absence makes the <i>heart</i> grow fonder</p>

<p id="ex"></p>

<script>
var x = document.getElementsByTagName("p");
document.getElementById("ex").innerHTML =
'The text in first paragraph is: ' + x[1].innerHTML;
</script>

</body>
</html>
```

Results:

JS Elements by Tag Name

A bad workman always blames his tools!

Absence makes the *heart* grow fonder

The text in first paragraph is: Absence makes the *heart* grow fonder

NOTE:

- The `getElementsByTagName()` is a method of the document or element object.

- The `getElementsByName()` accepts a tag name and returns a list of elements with the matching tag name.
- The `getElementsByName()` returns a live `HTMLCollection` of elements. The `HTMLCollection` is an array-like object.

Traversing elements :

Introduction to parentNode attribute

To get the parent node of a specified node in the DOM tree, you use the `parentNode` property:

let parent = node.parentNode;

The `parentNode` is read-only. The Document and Document Fragment nodes do not have a parent, therefore the `parentNode` will always be null.

If you create a new node but haven't attached it to the DOM tree, the `parentNode` of that node will also be null.

JavaScript parentNode example, See the following HTML document:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JavaScript parentNode</title>
</head>
<body>
  <div id="main">
    <p class="note">This is a note!</p>
  </div>

  <script>
    let note = document.querySelector('.note');
    console.log(note.parentNode);
  </script>
</body>
</html>
```


To get the next sibling of an element, you use the `nextElementSibling` attribute:

```
let nextSibling = currentNode.nextElementSibling;
```

The `nextElementSibling` returns null if the specified element is the first one in the list. The following example uses the `nextElementSibling` property to get the next sibling of the list item that has the current class:

```
let current = document.querySelector('.current');
```

```
let nextSibling = current.nextElementSibling;
```

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

To get the previous siblings of an element, you use the `previousElementSibling` attribute:

```
let current = document.querySelector('.current');
```

```
let prevSibling = currentNode.previousElementSibling;
```

The `previousElementSibling` property returns null if the current element is the first one in the list.

The following example uses the `previousElementSibling` property to get the previous siblings of the list item that has the current class:

```
let current = document.querySelector('.current');
```

```
let prevSiblings = current.previousElementSibling;
```

```
console.log(prevSiblings);
```

The `nextElementSibling` returns the next sibling of an element or null if the element is the last one in the list.

The `previousElementSibling` returns the previous sibling of an element or null if the element is the first one in the list.

To get all siblings of an element, you can use a helper function that utilizes the `nextElementSibling` property.

The `firstChild` and `lastChild` return the first and last child of a node, which can be any node type including text node, comment node, and element node.

The `firstElementChild` and `lastElementChild` return the first and last child Element node.

The `childNodes` returns a live `NodeList` of all child nodes of any node type of a specified node. The `children` return all child Element nodes of a specified node.

Manipulating Elements

The concept of DOM Manipulation can be broken down to CRUD, to make it more understandable. CRUD stands for Create, Read, Update and Delete. In reference to DOM, it refers to:

- 1. Creating DOM Nodes*
- 2. Reading the DOM Nodes*
- 3. Updating the DOM Nodes*
- 4. Deleting the DOM Nodes*

Reading DOM Nodes

There are many ways to read/access the DOM nodes. Some of the most important ones are:

querySelector

The `querySelector()` method (of the document object) returns the first element that matches a specified *CSS selector(s)* in the document.

Syntax:

```
document.querySelector('<Selector>');
```

selectors

A DOMString containing one or more selectors to match. This string must be a valid CSS selector string.

Return value

An HTML element object representing the first element in the document that matches the specified set of CSS selectors, or null is returned if there are no matches.

querySelectorAll

The `querySelectorAll()` method (of the document object) returns all elements that match a specified *CSS selector(s)* in the document.

Syntax:

```
document.querySelectorAll('<Selector>');
```

selectors

A DOMString containing one or more selectors to match against. This string must be a valid CSS selector string.

Return value

A non-live NodeList containing one Element object for each element that matches at least one of the specified selectors or an empty NodeList in case of no matches.

Updating DOM Nodes

Every node in the DOM has a set of properties and methods attached to it. These properties and methods are used to update the respective DOM nodes. To change the text node of a particular element node, `innerText` property of element node is used.

Syntax:

```
elementNode.innerText = 'Modified Text';
```

Deleting DOM Nodes

To delete a particular DOM node (in VanillaJS), we have to first access its parent node and from there delete the child.

Syntax:

```
elementNode.parentNode.removeChild(elementNode);
```

Creating DOM Nodes

Different types of DOM nodes are created using different syntaxes.

//creating a new element node

Syntax: **document.createElement('<HTML element>');**

//creating a new text node

Syntax: **document.createTextNode('New text content');**

Adding elements to DOM

The nodes that are created using JavaScript, need to be added to the DOM also, else there is no use of creating the node. Addition of a node to the DOM makes it visible in the browser. The newly created node is appended to an existing node. This is done by using `appendChild()`.

AppendChild

The `appendChild()` method adds a node to the end of the list of children of a specified parent node. If the given child is a reference to an existing node in the document, `appendChild()` moves it from its current position to the new position.

Syntax:

`elementNode.appendChild(elementNode);`

setAttribute()

Sets the value of an attribute on the specified element. If the attribute already exists, the value is updated; otherwise a new attribute is added with the specified name and value.

To get the current value of an attribute, use `getAttribute()`; to remove an attribute, call `removeAttribute()`.

Syntax:

`Element.setAttribute(name, value);`

name

Its specifying the name of the attribute whose value is to be set.

value

It's containing the value to assign to the attribute. Any non-string value specified is converted automatically into a string. Boolean attributes are considered to be true if they're present on the element at all, regardless of their actual value; as a rule, should specify the empty string ("") in value.

getAttribute()

The `getAttribute()` method of the `Element` interface returns the value of a specified attribute on the element. If the given attribute does not exist, the value returned will either be `null` or `""` i.e the empty string.

Syntax :

let attribute = element.getAttribute(attributeName);

attribute is a string containing the value of *attributeName*.

attributeName is the name of the attribute whose value you want to get.

CODE:

```
<!DOCTYPE html>
<html>
<head>
<style>
.proverb{
  color: indigo;
}
</style>
</head>
<body>

<h1 class="proverb">A chain is only as strong as its weakest link.</h1>
<p>One weak part will render the whole weak.</p>

<button onclick="myFunction()">Try it</button>

<p id="EX"></p>

<script>
function myFunction()
{
  var x = document.getElementsByTagName("H1")[0].getAttribute("class");
  document.getElementById("EX").innerHTML = x;
}
</script>
</body>
</html>
```

Results:

A chain is only as strong as its weakest link.

One weak part will render the whole weak.

Try it

When you click Try It, it displays the value of `getAttribute`.

A chain is only as strong as its weakest link.

One weak part will render the whole weak.

Try it

proverb

JavaScript innerHTML vs createElement

1) createElement is more performant

Suppose that you have a div element with the class container:

```
<div class="container"></div>
```

You can new elements to the div element by creating an element and appending it:

```
let div = document.querySelector('.container');
```

```
let p = document.createElement('p');
```

```
p.textContent = 'JS DOM';
```

```
div.appendChild(p);
```

You can also manipulate an element's HTML directly using innerHTML like this:

```
let div = document.querySelector('.container');
```

```
div.innerHTML += '<p>JS DOM</p>';
```

Using innerHTML is cleaner and shorter when you want to add attributes to the element:

```
let div = document.querySelector('.container');
```

```
div.innerHTML += '<p class="note">JS DOM</p>';
```

However, using the innerHTML causes the web browsers to reparse and recreate all DOM nodes inside the div element. Therefore, it is less efficient than creating a new element and appending to the div. In other words, creating a new element and appending it to the DOM tree provides better performance than the innerHTML.

2) createElement is more secure

As mentioned in the innerHTML tutorial, you should use it only when the data comes from a trusted source like a database.

If you set the contents that you have no control over to the innerHTML, the malicious code may be injected and executed.

3) Using Document Fragment for composing DOM Nodes

Assuming that you have a list of elements and you need in each iteration:

```
let div = document.querySelector('.container');
```

```
for (let i = 0; i < 1000; i++) {
```

```
    let p = document.createElement('p');
```

```
    p.textContent = `Paragraph ${i}`;
```



```
div.appendChild(p);  
  
}
```

This code results in recalculation of styles, painting, and layout every iteration. This is not very efficient.

To overcome this, you typically use a Document Fragment to compose DOM nodes and append it to the DOM tree:

```
let div = document.querySelector('.container');  
  
// compose DOM nodes  
  
let fragment = document.createDocumentFragment();  
  
for (let i = 0; i < 1000; i++) {  
  let p = document.createElement('p');  
  
  p.textContent = `Paragraph ${i}`;  
  
  fragment.appendChild(p);  
  
}  
  
// append the fragment to the DOM tree  
  
div.appendChild(fragment);
```

In this example, we composed the DOM nodes by using the Document Fragment object and append the fragment to the active DOM tree once at the end.

A document fragment does not link to the active DOM tree, therefore, it doesn't incur any performance.

Unit 2: HTML5, JQuery and Ajax

JavaScript events

An event is an action that occurs in the web browser, which the web browser feeds back to you so that you can respond to it.

For example, when users click a button on a webpage, you may want to respond to this click event by displaying a dialog box.

Each event may have an event handler which is a block of code that will execute when the event occurs.

An event handler is also known as an event listener. It listens to the event and executes when the event occurs.

Suppose you have a button with the id btn:

```
<button id="btn">Click Me!</button>
```

To define the code that will be executed when the button is clicked, you need to register an event handler using the `addEventListener()` method:

```
let btn = document.querySelector('#btn');  
  
function display() {  
  
  alert('It was clicked!');  
  
}  
  
btn.addEventListener('click',display);
```

How it works.

- First, select the button with the id btn by using the `querySelector()` method.

- Then, define a function called `display()` as an event handler.
- Finally, register an event handler using the `addEventListener()` so that when users click the button, the `display()` function will be executed.

A shorter way to register an event handler is to place all code in an anonymous function, like this:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function() {

    alert('It was clicked!');

});
```

Event object

To handle an event properly we need want to know what's happened when an event occurs. It's not just a "click" or a "keydown", but what were the pointer coordinates at the time and which key was pressed.

When an event happens, the browser creates an event object, puts details into it and passes it as an argument to the handler. A parameter specified with a name such as **event**, **evt**, or simply **e**. This is called the **event object**, and it is automatically passed to event handlers to provide extra features and information.

Here's an example of getting pointer coordinates from the event object:

CODE:

```
<!DOCTYPE html>
<html>
<head>
  <title>JS Event Demo</title>
</head>
<body>
  <input type="button" value="Click me" id="elem">
  <script>
    elem.onclick = function(event) {
      // show event type, element and coordinates of the click
      alert(event.type + " at " + event.currentTarget);
      alert("Coordinates: " + event.clientX + ":" + event.clientY);
    };
  </script>
```

Results:

Click me

This page says

click at [object HTMLInputElement]

OK

This page says

Coordinates: 32:21

OK

Some properties of event object:

event.type

Event type, here it's "click"

event.currentTarget

Element that handled the event. That's exactly the same as this, unless the handler is an arrow function, or its this is bound to something else, then we can get the element from event.currentTarget.

event.clientX / event.clientY

Window-relative coordinates of the cursor, for pointer events. When the event occurs, the web browser passed an Event object to the event handler:

```
let btn = document.querySelector('#btn');  
  
btn.addEventListener('click', function(event) {  
  
    console.log(event.type);  
  
});
```

Output:

'click'

event.target

A handler on a parent element can always get the details about where it actually happened.

The most deeply nested element that caused the event is called a target element, accessible as event.target.

Note the differences from this (=event.currentTarget):

event.target – is the “target” element that initiated the event, it doesn't change through the bubbling process.

this – is the “current” element, the one that has a currently running handler on it.

For instance, if we have a single handler `form.onclick`, then it can “catch” all clicks inside the form. No matter where the click happened, it bubbles up to `<form>` and runs the handler.

In `form.onclick` handler:

`this (=event.currentTarget)` is the `<form>` element, because the handler runs on it.

`event.target` is the actual element inside the form that was clicked.

The following table shows the most commonly-used properties and methods of the event object:

Property/Method	Description
bubbles	true if the event bubbles
cancelable	true if the default behavior of the event can be canceled
currentTarget	the current element on which the event is firing
defaultPrevented	return true if the <code>preventDefault()</code> has been called.
detail	more information about the event
eventPhase	1 for capturing phase, 2 for target, 3 for bubbling
preventDefault()	cancel the default behavior for the event. This method is only effective if the cancelable property is true
stopPropagation()	cancel any further event capturing or bubbling. This method only can be used if the bubbles property is true.
target	the target element of the event
type	the type of event that was fired

The event object is only accessible inside the event handler. Once all the event handlers have been executed, the event object is automatically destroyed.

preventDefault()

To prevent the default behavior of an event, you use the `preventDefault()` method.

For example, when you click a link, the browser navigates you to the URL specified in the href attribute:

```
<a href="https://www.google.com/">Google</a>
```

However, you can prevent this behaviour by using the `preventDefault()` method of the event object:

```
let link = document.querySelector('a');  
  
link.addEventListener('click',function(event) {  
  
    console.log('clicked');  
  
    event.preventDefault();  
  
});
```

Note:

The `preventDefault()` method does not stop the event from bubbling up the DOM. And an event can be canceled when its cancelable property is true.

stopPropagation()

The `stopPropagation()` method immediately stops the flow of an event through the DOM tree. However, it does not stop the browser's default behavior.

Example:

```
let btn = document.querySelector('#btn');  
  
btn.addEventListener('click', function(event) {  
  
    console.log('The button was clicked!');  
  
    event.stopPropagation();  
  
});  
  
document.body.addEventListener('click',function(event) {  
  
    console.log('The body was clicked!');  
  
});
```

Without the `stopPropagation()` method, you would see two messages on the Console window.

The click event never reaches the body because the `stopPropagation()` was called on the click event handler of the button.

When an event occurs, you can create an event handler which is a piece of code that will execute to respond to that event. An event handler is also known as an event listener. It listens to the event and responds accordingly to the event fires.

An event listener is a function with an explicit name if it is reusable or an anonymous function in case it is used one time.

An event can be handled by one or multiple event handlers. If an event has multiple event handlers, all the event handlers will be executed when the event is fired.

There are three ways to assign event handlers.

1) HTML event handler attributes

Event handlers typically have names that begin with on, for example, the event handler for the click event is onclick.

To assign an event handler to an event associated with an HTML element, you can use an HTML attribute with the name of the event handler. For example, to execute some code when a button is clicked, you use the following:

```
<input type="button" value="Save" onclick="alert('Clicked!')">
```

In this case, when the button is clicked, the alert box is shown.

When you assign JavaScript code as the value of the onclick attribute, you need to escape the HTML characters such as ampersand (&), double quotes ("), less than (<), etc., or you will get a syntax error.

An event handler defined in the HTML can call a function defined in a script. For example:

```
<script>  
  
function showAlert() {  
  
    alert('Clicked!');}  
  
</script>
```

```
<input type="button" value="Save" onclick="showAlert()">
```

In this example, the button calls the showAlert() function when it is clicked. The showAlert() is a function defined in a separate <script> element, and could be placed in an external JavaScript file.

NOTE:

The following are some important points when you use the event handlers as attributes of the HTML element:

First, the code in the event handler can access the event object without explicitly defining it:

```
<input type="button" value="Save" onclick="alert(event.type)">
```

Second, the `this` value inside the event handler is equivalent to the event's target element:

```
<input type="button" value="Save" onclick="alert(this.value)">
```

Third, the event handler can access the element's properties, for example:

```
<input type="button" value="Save" onclick="alert(value)">
```

Disadvantages of using HTML event handler attributes

Assigning event handlers using HTML event handler attributes are considered as bad practices and should be avoided as much as possible because of the following reasons:

1. First, the event handler code is mixed with the HTML code, which will make the code more difficult to maintain and extend.
2. Second, it is a timing issue. If the element is loaded fully before the JavaScript code, users can start interacting with the element on the webpage which will cause an error.

For example, suppose that the following `showAlert()` function is defined in an external JavaScript file:

```
<input type="button" value="Save" onclick="showAlert()">
```

And when the page is loaded fully and the JavaScript has not been loaded, the `showAlert()` function is undefined. If users click the button at this moment, an error will occur.

2) DOM Level 0 event handlers

Each element has event handler properties such as onclick. To assign an event handler, you set the property to a function as shown in the example:

```
let btn = document.querySelector('#btn');  
  
btn.onclick = function() {  
  
    alert('Clicked!');  
  
};
```

In this case, the anonymous function becomes the method of the button element. Therefore, the this value is equivalent to the element. And you can access the element's properties inside the event handler:

```
let btn = document.querySelector('#btn');  
  
btn.onclick = function() {  
  
    alert(this.id);  
  
};
```

Output:

btn

By using the this value inside the event handler, you can access the element's properties and methods.

To remove the event handler, you set the value of the event handler property to null:

```
btn.onclick = null;
```

The DOM Level 0 event handlers are still being used widely because of its simplicity and cross-browser support.

3) DOM Level 2 event handlers

DOM Level 2 Event Handlers provide two main methods for dealing with the registering/deregistering event listeners:

addEventListener() – register an event handler

removeEventListener() – remove an event handler

These methods are available in all DOM nodes.

The addEventListener() method

The `addEventListener()` method accepts three arguments: an event name, an event handler function, and a Boolean value that instructs the method to call the event handler during the capture phase (`true`) or during the bubble phase (`false`).

For example:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function(event) {

    alert(event.type); // click

});
```

It is possible to add multiple event handlers to handle a single event, like this:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function(event) {

    alert(event.type); // click

});

btn.addEventListener('click',function(event) {

    alert('Clicked!');
```

```
});
```

The removeEventListener() method

The removeEventListener() removes an event listener that was added via the addEventListener(). However, you need to pass the same arguments as were passed to the addEventListener(). For example:

```
let btn = document.querySelector('#btn');

// add the event listener

let showAlert = function() {

    alert('Clicked!');

};

btn.addEventListener('click', showAlert);

// remove the event listener

btn.removeEventListener('click', showAlert);
```

Using an anonymous event listener as the following will not work:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function() {

    alert('Clicked!');

});    // won't work

btn.removeEventListener('click', function() {

    alert('Clicked!');

});
```

Accessing the element: this

The value of this inside a handler is the element. The one which has the handler on it. In the code below button shows its contents using this.innerHTML:

```
<button onclick="alert(this.innerHTML)">Click me</button>
```

We can set an existing function as a handler, the function should be assigned as sayThanks, not sayThanks().

If we add parentheses, then sayThanks() becomes is a function call. So the last line actually takes the result of the function execution, that is undefined (as the function returns nothing), and assigns it to onclick. That doesn't work.

Don't use setAttribute for handlers.

Example:

```
// a click on <body> will generate errors,  
  
// because attributes are always strings, function becomes a string
```

```
document.body.setAttribute('onclick', function() { alert(1) });
```

JavaScript Event Delegation

Suppose that you have the following menu:

```
<ul id="menu">  
  
    <li><a id="company"> company </a></li>  
  
    <li><a id="Employee"> Employee </a></li>  
  
    <li><a id="report">report</a></li>  
  
</ul>
```

To handle the click event of each menu item, you may add the corresponding click event handlers:

```
let company = document.querySelector('# company');  
  
company.addEventListener(company,(event) => {  
  
console.log('company menu item was clicked');  
  
});  
  
let Employee = document.querySelector('# Employee);  
  
Employee.addEventListener(' Employee ',(event) => {  
  
console.log('Employee menu item was clicked');  
  
});  
  
let report = document.querySelector('#report');  
  
report.addEventListener('report',(event) => {  
  
console.log('Report menu item was clicked');  
  
});
```

In JavaScript, if you have a large number of event handlers on a page, these event handlers will directly impact the performance because of the following reasons:

- First, each event handler is a function which is also an object that takes up memory. The more objects in the memory, the slower the performance.
- Second, it takes time to assign all the event handlers, which causes a delay in the interactivity of the page.

To solve this issue, you can leverage the event bubbling.

Instead of having multiple event handlers, you can assign a single event handler to handle all the click events:

```
let menu = document.querySelector('#menu');
menu.addEventListener('click', (event) => {
  ...let target = event.target;
  ...switch(target.id) {
  ....case ' company ':
  .....console.log(' company menu item was clicked');
  .....break;
  ....case ' Employee ':
  .....console.log(' Employee menu item was clicked');
  .....break;
  ....case 'report':
  .....console.log('Report menu item was clicked');
  .....break;
  ....}
  ...});
```

Explanation:

When you click any <a> element inside the element with the id menu, the click event bubbles to the parent element which is the element. So instead of handling the click event of the individual <a> element, you can capture the click event at the parent element.

In the click event listener, you can access the target property which references the element that dispatches the event. To get the id of the element that the event actually fires, you use the target.id property.

Once having the id of the element that fires the click event, you can have code that handles the event correspondingly.

The way that we handle the too-many-event-handlers problem is called the **event delegation**.

The event delegation refers to the technique of leveraging event bubbling to handle events at a higher level in the DOM than the element on which the event originated.

JavaScript event delegation benefits

When it is possible, you can have a single event handler on the document that will handle all the events of a particular type. By doing this, you gain the following benefits:

- Less memory usage, better performance.
- Less time required to set up event handlers on the page.
- The document object is available immediately. As long as the element is rendered, it can start functioning correctly without delay. You don't need to wait for the DOMContentLoaded or load events.

Delegation limitations:

- First, the event must be bubbling. Some events do not bubble. Also, low-level handlers should not use event.stopPropagation().
- Second, the delegation may add CPU load, because the container-level handler reacts on events in any place of the container, no matter whether they interest us or not. But usually the load is negligible, so we don't take it into account.

JavaScript custom events

The following function highlights an element by changing its background color to yellow:

```
function highlight(elem) {  
  
    const bgColor = 'yellow';  
  
    elem.style.backgroundColor = bgColor;  
  
}
```

To execute a piece of code after highlighting the element, you may come up with a callback:

```
function highlight(elem, callback) {  
  
const bgColor = 'yellow';  
  
elem.style.backgroundColor = bgColor;  
  
if(callback && typeof callback === 'function') {  
  
callback(elem);  
  
}  
  
}
```

The following calls the highlight() function and adds a border to a <div> element:

CODE :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>JS Custom Event Demo</title>
</head>
<body>
  <div class="note">JS Custom Event Demo</div>
  <script>
    <!--function highlight(elem, callback) {
    <!--const bgColor = 'yellow';
    <!--elem.style.backgroundColor = bgColor;

    <!--if (callback && typeof callback === 'function') {
    <!--callback(elem);
    <!--}
    <!--}
let note = document.querySelector('.note');
    <!--function addBorder(elem) {
    <!--elem.style.border = "solid 1px red";
    <!--}
    <!--highlight(note, addBorder);
  </script>
</body>
</html>
```

Results:

JS Custom Event Demo

To make the code more flexible, you can use the custom event.

Creating JavaScript custom events

To create a custom event, you use the CustomEvent() constructor:

let event = new CustomEvent(eventType, options);

The CustomEvent() has two parameters:

- The eventType is a string that represents the name of the event.

- The options is an object has the detail property that contains any custom information about the event.

The following example shows how to create a new custom event called highlight:

```
let event = new CustomEvent('highlight', {  
    detail: {backgroundColor: 'yellow'}});
```

Dispatching JavaScript custom events

After creating a custom event, you need to attach the event to an element and trigger it by using the `dispatchEvent()` method:

```
element.dispatchEvent(event);
```

NOTE :

- Use the `CustomEvent()` constructor to create a custom event and `dispatchEvent()` to trigger the event.
- The custom events allow you to decouple the code that you want to execute after another piece of code completes.
- For example, you can separate the event listeners in a separate script. In addition, you can have multiple event listeners to the same custom event.

JavaScript page load events

When you open a page, the following events occur in sequence:

- ***DOMContentLoaded*** – the browser fully loaded HTML and completed building the DOM tree. However, it hasn't loaded external resources like stylesheets and images. In this event, you can start selecting DOM nodes or initialize the interface.
- ***load*** – the browser fully loaded the HTML and also external resources like images and stylesheets.

When you leave the page, the following events fire in sequence:

- ***beforeunload*** – fires before the page and resources are unloaded. You can use this event to show a confirmation dialog to confirm if you really want to leave the page. By doing this, you can prevent data loss in case you are filling out a form and accidentally click a link to navigate to another page.
- ***unload*** – fires when the page has completely unloaded. You can use this event to send the analytic data or to clean up resources.

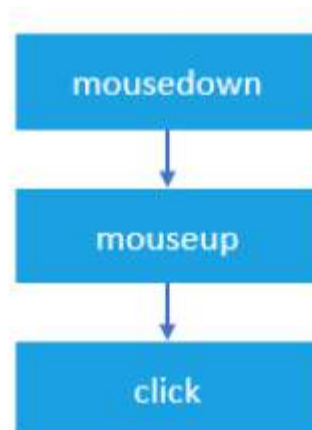
JavaScript mouse events

Mouse events fire when you use the mouse to interact with the elements on the page. DOM Level 3 events define nine mouse events.

mousedown, mouseup, and click

When you click an element, there are no less than three mouse events fire in the following sequence:

- The mousedown fires when you depress the mouse button on the element.
- The mouseup fires when you release the mouse button on the element.
- The click fires when one mousedown and one mouseup detected on the element.



If you depress the mouse button on an element and move your mouse off the element, and then release the mouse button. The only mousedown event fires on the element.

Likewise, if you depress the mouse button, and move the mouse over the element, and release the mouse button, the only mouseup event fires on the element.

In both cases, the click event never fires.

dblclick

In practice, you rarely use the dblclick event. The dblclick event fires when you double click over an element.

It takes two click events to cause a dblclick event to fire. The dblclick event has four events fired in the following order:

- mousedown
- mouseup
- click
- mousedown
- mouseup
- click
- dblclick

As you can see, the click events always take place before the dblclick event. If you register both click and dblclick event handlers on the same element, you will not know exactly what user actually has clicked or double-clicked the element.

mousemove

The *mousemove* event fires repeatedly when you move the mouse cursor around an element. Even when you move the mouse one pixel, the *mousemove* event still fires. It will cause the page slow, therefore, you only register *mousemove* event handler only when you need it and immediately remove the event handler as soon as it is no longer used, like this:

```
element.onmousemove = mouseMoveEventHandler;
```

```
// ...
```

```
// later, no longer use
```

```
element.onmousemove = null;
```

mouseover / mouseout

The *mouseover* fires when the mouse cursor is outside of the element and then move to inside the boundaries of the element.

The *mouseout* fires when the mouse cursor is over an element and then move another element.

mouseenter / mouseleave

The *mouseenter* fires when the mouse cursor is outside of an element and then moves to inside the boundaries of the element.

The *mouseleave* fires when the mouse cursor is over an element and then moves to the outside of the element's boundaries.

Both *mouseenter* and *mouseleave* does not bubble and does not fire when the mouse cursor moves over descendant elements.

Registering mouse event handlers

To register a mouse event, you use these steps:

First, select the element by using `querySelector()` or `getElementById()` method.

Then, register the mouse event using the `addEventListener()` method.

For example, suppose that you have the following button:

```
<button id="btn">Click Me!</button>
```

To register a mouse click event handler, you use the following code:

```
let btn = document.querySelector('#btn');  
  
btn.addEventListener('click',(event) => {  
  
    console.log('clicked');});
```

or

you can assign a mouse event handler to the element's property:

```
let btn = document.querySelector('#btn');  
  
btn.onclick = (event) => {  
  
    console.log('clicked');};
```

In legacy systems, you may find that the event handler is assigned in the HTML attribute of the element:

```
<button id="btn" onclick="console.log('clicked')">Click Me!</button>
```

It's a good to always use the `addEventListener()` to register a mouse event handler.

Detecting mouse buttons

The event object passed to the mouse event handler has a property called `button` that indicates which mouse button was pressed on the mouse to trigger the event.

The mouse button is represented by a number:

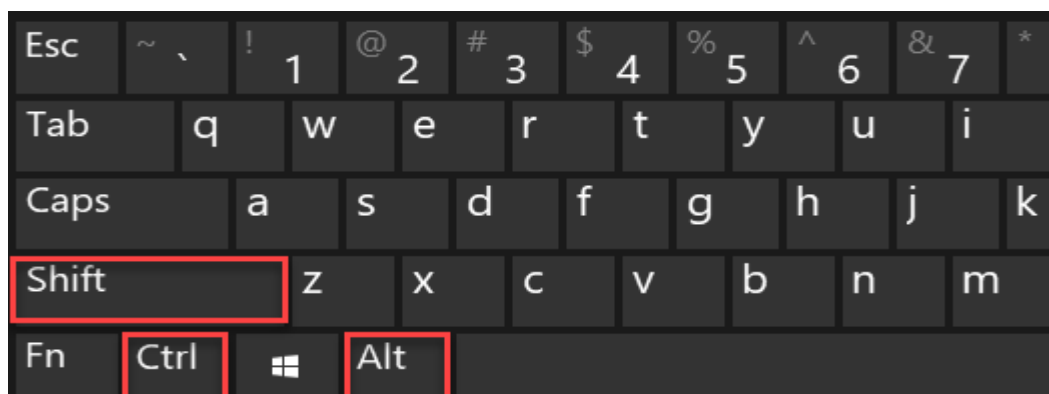
- 0: the main mouse button pressed, usually the left button.
- 1: the auxiliary button pressed, usually the middle button or the wheel button.
- 2: the secondary button pressed, usually the right button.
- 3: the fourth button pressed, usually the Browser Back button.
- 4: the fifth button pressed, usually the Browser Forward button.



Modifier keys

When you click an element, you may press one or more modifier keys:

Shift, Ctrl, Alt, and Meta.



Note:

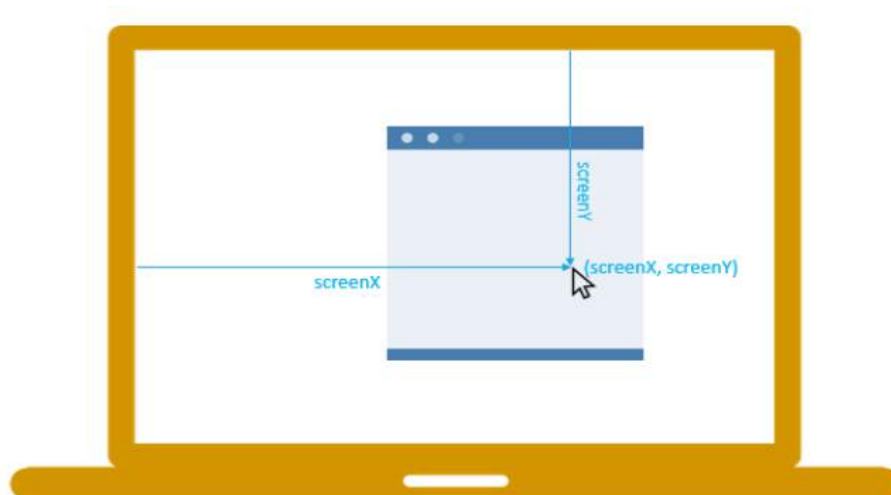
The Meta key is the Windows key on Windows keyboards and the Command key on Apple keyboard.

To detect if these modifier keys have been pressed, you can use the event object passed to the mouse event handler.

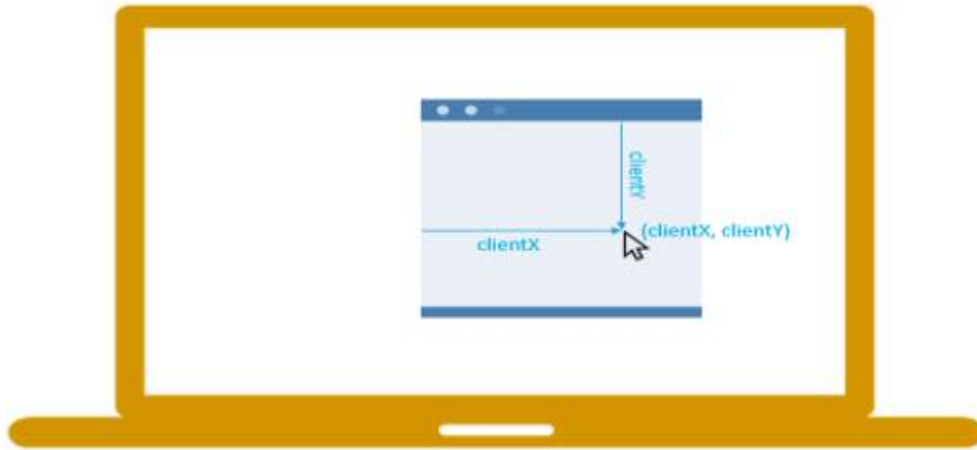
The event object has four Boolean properties, where each is set to true if the key is being held down or false if the key is not pressed.

Screen Coordinates

The *screenX* and *screenY* properties of the event passed to the mouse event handler return the screen coordinates of the location of the mouse in relation to the entire screen.



On the other hand, the *clientX* and *clientY* properties provide the horizontal and vertical coordinates within the application's client area at which the mouse event occurred:



NOTES:

- Use `addEventListener()` method to register a mouse event handler.
- The `event.button` indicates which mouse button was pressed to trigger the mouse event.
- The modifier keys: `alt`, `shift`, `ctrl`, and `meta` (Mac) can be obtained via properties of the event object passed to the mouse event handler.
- The `screenX` and `screenY` properties return the horizontal and vertical coordinates of the mouse pointer in screen coordinates.
- The `clientX` and `clientY` properties of the event object return horizontal and vertical coordinates within the application's client area at which the mouse event occurred.

keyboard events

When you interact with the keyboard, the keyboard events are fired. There are three main keyboard events:

- ***keydown*** – fires when you press a key on the keyboard and it fires repeatedly while you holding down the key.
- ***keyup*** – fires when you release a key on the keyboard.

- **keypress** – fires when you press a character keyboard like a,b, or c, not the left arrow key, home, or end keyboard, ... The keypress also fires repeatedly while you hold down the key on the keyboard.

The keyboard events typically fire on the text box, through all elements support them.

When you press a character key once on the keyboard, three keyboard events are fired in the following order:

- keydown
- keypress
- keyup

Both keydown and keypress events are fired before any change made to the text box, whereas the keyup event fires after the changes have made to the text box. If you hold down a character key, the keydown and keypress are fired repeatedly until you release the key.

When you press a non-character key, the keydown event is fired first followed by the keyup event. If you hold down the non-character key, the keydown is fired repeatedly until you release the key.

Handling keyboard events

To handle a keyboard event, you follow these steps:

First, select the element on which the keyboard event will fire. Typically, it is a text box.

Then, use the `element.addEventListener()` to register an event handler.

Suppose that you have the following text box with the id message:

<input type="text" id="message">

The following example illustrates how to register keyboard event listeners:

```
let msg = document.getElementById('#message');  
  
msg.addEventListener('keydown', (event) => {  
  
    // handle keydown });  
  
msg.addEventListener('keypress', (event) => {  
  
    // handle keypress });  
  
msg.addEventListener('keyup', (event) => {  
  
    // handle keyup });
```

If you press a character key, all three event handlers will be called.

The keyboard event properties

The keyboard event has two important properties: key and code. The key property returns the character that has been pressed whereas the code property returns the physical key code.

For example, if you press the z character key, the event.key returns z and event.code returns KeyZ.

NOTE :

- When you press a character key on the keyboard, the keydown, keypress, and keyup events are fired sequentially. However, if you press a non-character key, only the keydown and keyup events are fired.
- The keyboard event object has two important properties: key and code properties that allow you to detect which key has been pressed.
- The key property returns the value of the key pressed while the code represents a physical key on the keyboard.

JavaScript focus events

The focus events fire when an element receives or loses focus. These are the two main focus events:

- *focus* fires when an element has received focus.
- *blur* fires when an element has lost focus.
- The *focusin* and *focusout* fire at the same time as focus and blur, however, they bubble while the focus and blur do not.

The following elements are focusable:

- The window gains focus when you bring it forward by using Alt+Tab or clicking on it and loses focus when you send it back.
- Links when you use a mouse or a keyboard.
- Form fields like input text when you use a keyboard or a mouse.
- Elements with tabindex, also when you use a keyboard or a mouse

Unit 2: HTML5, JQuery And Ajax

Handling Events in JavaScript

Event flow

Assuming that you have the following HTML document:

```
<!DOCTYPE html>
<html>
<head>
  <title>JS Event Demo</title>
</head>
<body>
  <div id="container">
    <button id='btn'>Click Me!</button>
  </div>
</body>
```

When you click the button, you're clicking not only the button but also the button's container, the div, and the whole webpage.

Event flow explains the order in which events are received on the page from the element where the event occurs and propagated through the DOM tree.

There are two main event models: **event bubbling** and **event capturing**.

Event bubbling

In the event bubbling model, an event starts at the most specific element and then flows upward toward the least specific element (the document or even window).

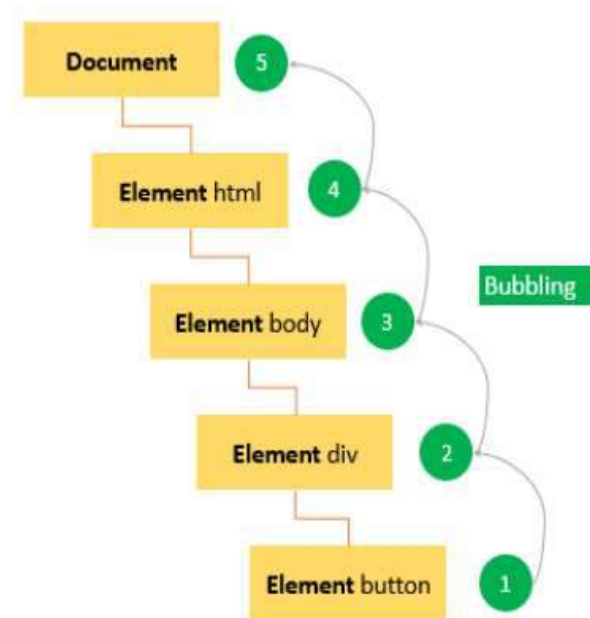
When you click the button, the click event occurs in the following order:

- button
- div with the id container

- body
- html
- document

The click event first occurs on the button which is the element that was clicked. Then the click event goes up the DOM tree, firing on each node along its way until it reaches the document object. Modern web browsers bubble the event up to the window object.

The following picture illustrates the event bubbling effect when users click the button:



Event capturing

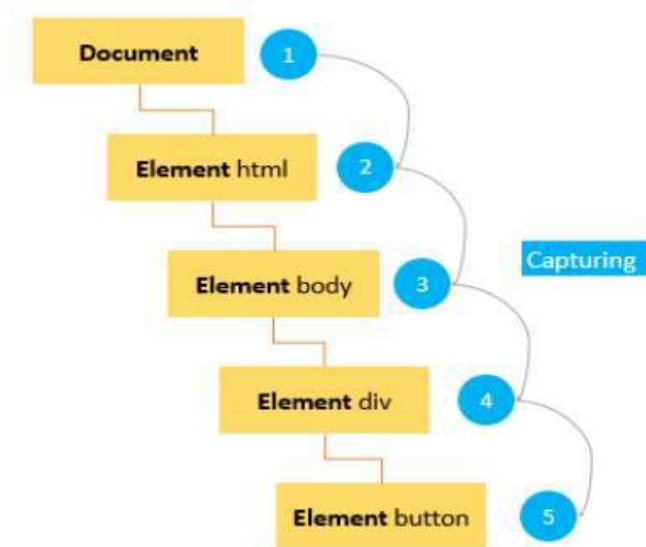
In the event capturing model, an event starts at the least specific element and flows downward toward the most specific element.

When you click the button, the click event occurs in the following order:

- document
- html

- body
- div with the id container
- button

The following picture illustrates the event capturing effect:

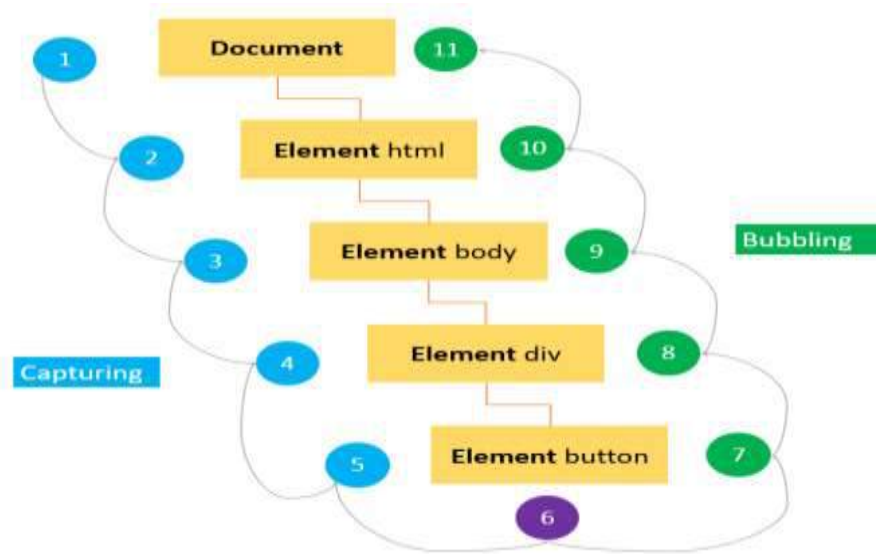


DOM Level 2 Event flow

DOM level 2 events specify that event flow has three phases:

1. First, event capturing occurs, which provides the opportunity to intercept the event.
2. Then, the actual target receives the event.
3. Finally, event bubbling occurs, which allows a final response to the event.

The following picture illustrates the DOM Level 2 event model when users click the button:



Submit & Reset Buttons

Buttons are defined by changing the input element's type attribute. There are two different kinds of buttons – the submit button and the reset button. Submit and Reset buttons help the user tell the browser what to do with a form on the website.

The INPUT element defines an input field. When specify "submit" (or "reset") for the type attribute of this element, a submit button (or a reset button) is created.

```
<input type="submit" value="Submit">
```

```
<input type="reset" value="Reset">
```

Attribute	Value	Explanation
type=" "	submit	the type of input field submit : creates a submit button reset : creates a reset button
	reset	
name=" "	button	a unique name for the button

	name	
value=" "	button text	the text displayed on the button

type="submit"

Creates a submit button on the form.

When this button is clicked, the form data is submitted to the server.

type="reset"

Creates a reset button on the form.

When this button is clicked, the input is reset.

name=""

The button name is used to identify the clicked submit button.

value=""

Value is the text displayed on the button.

CODE:

```
<form action="url-to-formmail-provided-by-your-ISP" method="post">
  <input type="text" name="firstname" /> <br /></br>
  Surname: <input type="text" name="surname" /><br /></br>
  <input type="reset" value="Clear form" />
  <input type="submit" value="Submit now" />
</form>
```

OUTPUT:

First name:

Surname:

Unit 2: HTML5, JQuery and Ajax

HTML5 <audio> tag

It is used to play audio in html pages. It takes the below basic format in its simplest form:

```
<audio src="my_music.mp3" controls></audio>
```

With the above structure, when the html page loads the page requests for the audio file listed in the "src" attribute and the "controls" attribute displays the browser default audio player for controlling playback.

CODE:

```
<!doctype html>-
<html lang="en">-
<head>-
  <meta charset="UTF-8">
  <title>Intro to Audio</title>
</head>
<body>-
  <audio src="sample.mp3" controls></audio>-
</body>-
</html>-
```

Results:



The <audio> tag comes with many inline global attributes that help in modifying its behaviour. Some of the attributes are:

1. autoplay
2. buffered

3. controls
4. loop
5. muted
6. played
7. preload
8. src
9. Volume

Some of the attributes, values and description are given below:

Attribute	Value	Description
autoplay	autoplay	Specifies that the audio will start playing as soon as it is ready.
controls	controls	Specifies that controls will be displayed, such as a play button.
loop	loop	Specifies that the audio will start playing again (looping) when it reaches the end.
preload	preload	Specifies that the audio will be loaded at page load, and ready to run. Ignored if autoplay is present.
src	url	Specifies the URL of the audio to play.

HTML 5 <video> Tag

The HTML 5 <video> tag is used to specify video on an HTML document. For example, it can be embed in a music video on web page for visitors to listen to and watch. The <video> tag was introduced in HTML 5.

The HTML 5 <video> tag accepts attributes that specify how the video should be played. Attributes include preload, autoplay, loop and more.

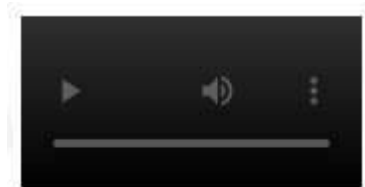
Any content between the opening and closing <video> tags is fallback content. This content is displayed only by browsers that don't support the <video> tag.

Attribute	Value	Description
audio	muted	Defining the default state of the the audio. Currently, only "muted" is allowed
autoplay	autoplay	If present, then the video will start playing as soon as it is ready
controls	controls	If present, controls will be displayed, such as a play button
height	<i>pixels</i>	Sets the height of the video player
loop	loop	If present, the video will start over again, every time it is finished
poster	<i>url</i>	Specifies the URL of an image representing the video
preload	preload	If present, the video will be loaded at page load, and ready to run. Ignored if "autoplay" is present
src	<i>url</i>	The URL of the video to play
width	<i>pixels</i>	Sets the width of the video player

CODE:

```
<video src="/video/pass-countdown.ogg" width="170" height="85" controls>
<p>If you are reading this, it is because your browser does not support the HTML5 video element.</p>
</video>
```

Results:



HTML 5 <progress> Tag

The <progress> element is used to create a progress bar to serve as a visual demonstration of progress towards the completion of task or goal. The max and value attributes are used to define how much progress (value) has been made towards task completion (max).

It can be indeterminate progress bar, which can be either in the form of spinning wheel or a horizontal bar. In this mode, the bar only shows cyclic movements and do not provide the exact progress indication. This mode is usually used at the time when the length of the time is not known.



CODE:

```
<progress value="33" max="100"></progress>
```

Results:



Reference Link for styling progress elements:

<https://css-tricks.com/html5-progress-element/>

Unit 2: HTML5, JQuery and Ajax

HTML 5 <canvas> Tag

The Canvas API provides a means for drawing graphics via JavaScript and the HTML <canvas> element. It can be used for animation, game graphics, data visualization, photo manipulation, and real-time video processing. Canvas allows you to render graphics powered by JavaScript. Some of the Canvas context methods are following:

Method	Description
fillRect(x, y, width, height)	Draws a filled rectangle
strokeRect(x, y, width, height)	Draws a rectangular outline
clearRect(x, y, width, height)	Clears the specified rectangular area, making it fully transparent
moveTo(x, y)	Moves the pen to the coordinates specified by x and y
lineTo(x, y)	Draws a line from the current drawing position to the position specified by x and y
arc(x, y, r, sAngle, eAngle, anticlockwise)	Draws an arc centered at (x, y) with radius r starting at sAngle and ending at eAngle going anticlockwise (defaulting to clockwise).
arcTo(x1, y1, x2, y2, radius)	Draws an arc with the given control points and radius, connected to the previous point by a straight line

CODE:

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
  -let canvas = document.querySelector("canvas");
  -let context = canvas.getContext("2d");
  -context.fillStyle = "red";
  -context.fillRect(10, 10, 100, 50);
</script>
```

Results:

Before canvas.



After canvas.

HTML <svg>Tag

The `svg` element is a container that defines a new coordinate system and viewport. It is used as the outermost element of SVG documents, but it can also be used to embed a SVG fragment inside an SVG or HTML document.

The `xmlns` attribute changes an element (and its children) to a different XML namespace. This namespace, identified by a URL, specifies the dialect that we are currently speaking. The `<circle>` and `<rect>` tags, which do not exist in HTML, do have a meaning in SVG—they draw shapes using the style and position specified by their attributes.

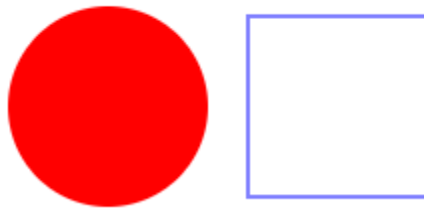
These tags create DOM elements, just like HTML tags, that scripts can interact with.

CODE:

```
<p>Normal HTML here.</p>  
<svg xmlns="http://www.w3.org/2000/svg">  
  <circle r="50" cx="50" cy="50" fill="red"/>  
  <rect x="120" y="5" width="90" height="90"  
    stroke="blue" fill="none"/>  
</svg>
```

Results:

Normal HTML here.



Unit 2: HTML5, JQuery and Ajax

HTML5 Geolocation API:

The Geolocation API of HTML5 helps in identifying the user's location, which can be used to provide location specific information or route navigation details to the user. There are many techniques used to identify the location of the user. The Geolocation API protects the user's privacy by mandating that the user permission should be sought and obtained before sending the location information of the user to any website. So the user will be prompted with a popover or dialog requesting for the user's permission to share the location information. The user can accept or deny the request.

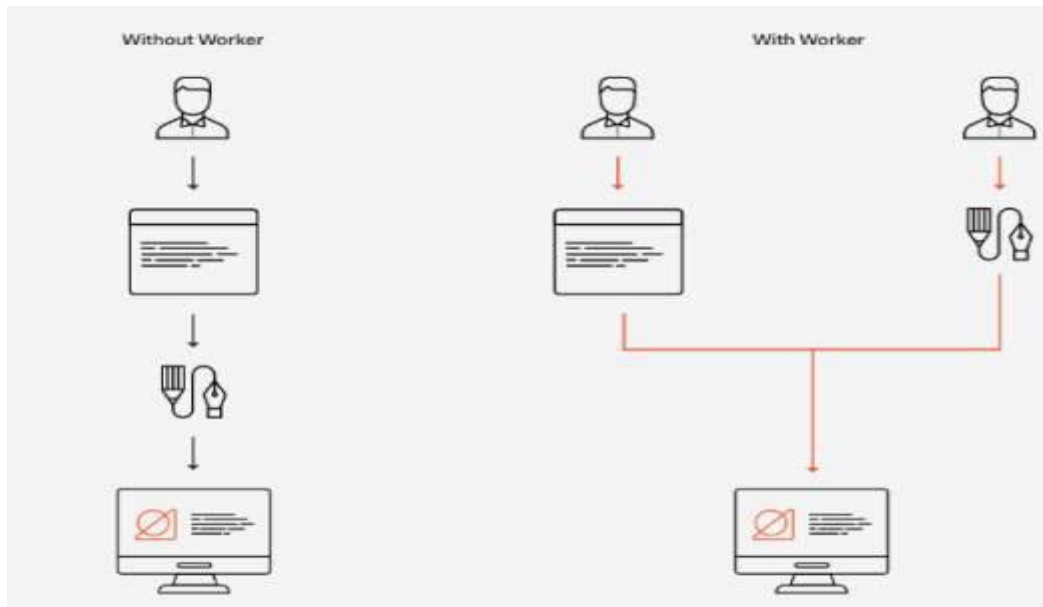
The current location of the user can be obtained using the `getCurrentPosition` function of the `navigator.geolocation` object. This function accepts three parameters – Success callback function, Error callback function and position options. If the location data is fetched successfully, the success callback function will be invoked with the obtained position object as its input parameter. Otherwise, the error callback function will be invoked with the error object as its input parameter.

Web Workers

Web Workers are a simple means for web content to run scripts in background threads. The worker thread can perform tasks without interfering with the user interface. In addition, they can perform I/O using `XMLHttpRequest` (although the `responseXML` and `channel` attributes are always null). Once created, a worker can send messages to the JavaScript code that created it by posting messages to an event handler specified by that code (and vice versa).

Using web workers in HTML5 allows you to prevent the execution of bigger tasks from freezing up your web page. A web worker performs the job in the

background, independent of other scripts and thus not affecting their performance. The process is also called threading, i.e., separating the tasks into multiple parallel threads. During the time, the user can browse normally, as the page stays fully responsive.



TYPES OF WEB WORKERS

Dedicated Workers

- Dedicated Web Workers are instantiated by the main process and can only communicate with it.
- A dedicated worker is only accessible by the script that called it.

Shared Workers

- Shared workers can be reached by all processes running on the same origin (different browser tabs, iframes or other shared workers).
- A shared worker is accessible by multiple scripts, similar to the basic

dedicated worker, except that it has two functions available handled by different script files

Workers are created and executed in one of the main program files with their code housed in a separate file. A worker is built using the Worker constructor method, which takes a parameter of worker.js, the JavaScript file storing the worker code.

Using .postMessage(), the parent thread can communicate messages to its workers. .postMessage() is a cross-origin API that can transmit primitive data and JSON structures, but not functions.

The parent code may also have a callback function that listens for a response from the worker confirming its work is complete in order for it to enact another action. As in the example below, the callback function will contain a target, which identifies the worker ('message'), and data, or the message posted by the worker.

This is main.js.

```
let worker = new Worker('worker.js');

worker.postMessage("Hello World");

worker.addEventListener('message', function(e) {
    console.log('Worker said: ', e.data);
}, false);
```

This is worker.js

```
self.addEventListener('message', function(e) {  
    self.postMessage(e.data);  
}, false);
```

Meanwhile, the worker, living in its own file, stands by with an eventListener waiting to be called. When it receives the message event from the parent code, it likewise communicates its response via the postMessage method.

HTML5 File API (Other Features in HTML5 – for reference only)

The HTML5 file API enables JavaScript inside HTML5 pages to load and process files from the local file system. Via the HTML5 file API it is possible for JavaScript to process a file locally, e.g. compress, encode or encrypt it, or upload the file in smaller chunks. Of course the HTML5 file API raises some security concerns.

Core Objects of HTML5 File API

The HTML5 file API contains the following core objects:

- FileList
- File
- Blob
- FileReader

The File object represents a file in the local file system.

The FileList object represents a list of files in the local file system. For instance, a list of files inside a directory.

The Blob object represents a Binary Large Object (BLOB) which is used to hold the contents of a single file from the local file system.

Unit 2: HTML5, JQuery and Ajax

jQuery

jQuery is a JavaScript Library. jQuery simplifies JavaScript programming. jQuery is a lightweight, "write less, do more", and JavaScript library. The purpose of jQuery is to make it much easier to use JavaScript on website. jQuery was originally released in January 2006 at BarCampNYC by John Resig.

jQuery is a JavaScript library that allows web developers to add extra functionality to their websites. It is open source and provided for free under the MIT license. In recent years, jQuery has become the most popular JavaScript library used in web development.

With jQuery you select (query) HTML elements and perform "actions" on them. Basic syntax is:

`$(selector).action ()`

- A \$ sign to define/ access jQuery.
- “\$()” access an element in current html document.

A (selector) to "query (or find)" HTML elements.

A jQuery action () to be performed on the element(s).

Examples:

- `$("p").hide()` - hides all elements.
- `$("#test").hide()` - hides the element with id="test".
- `$(".test").hide()` - hides all elements with class="test".

JavaScript vs. jQuery

Let us try to understand the difference between JavaScript and jquery.

Example 1 - Hide an element with id "textbox"

//JavaScript

```
document.getElementById(textbox).style.display = "none";
```

//jQuery

```
$("#textbox").hide();
```

Example 2 - Create a <h1> tag with "my text"

//JavaScript

```
var h1 = document.CreateElement("h1");
```

```
h1.innerHTML = "my text";
```

```
document.getElementsByTagName(body)[0].appendChild(h1);
```

//jQuery

```
$(body).append( $("<h1/>").html("my text") );
```

The Document Ready Event:

```
$(document).ready(function(){
```

```
    // jQuery methods go here...
```

```
})
```

This is to prevent any jQuery code from running before the document is finished loading (is ready).

- Trying to hide an element that is not created yet.
- Trying to get the size of an image that is not loaded yet

Alternate Syntax :

```
$(function){  
  
    // jQuery methods go here...  
  
    }
```

jquery Selectors:

jQuery selectors allow you to select and manipulate HTML. With jQuery selectors you can find elements based on their id, classes, types, attributes, values of attributes and much more. It's based on the existing CSS Selectors and in addition, it has some own custom selectors. All type of selectors in jQuery, start with the dollar sign and parentheses: \$().

Types of jquery selectors .

- Element selector Id
- (#) selector Class
- (.) selector

Element Selector:

The jQuery element selector selects elements based on their tag names.

Example:

```
$(document).ready(function()  
{  
    $("#button").click(function()  
    {  
        $("p").hide();  
    });  
});
```

Id (#) Selector :

The jQuery #id selector uses the id attribute of an HTML tag to find the specific element.

Example :

```
$(document).ready(function()
{
    $('#button').click(function()
    {
        $('#test').hide();
    });
});
```

Class (.) Selector:

The jQuery class selector finds elements with a specific class.

Example :

```
$(document).ready(function()
{
    $('#button').click(function()
    {
        $('.test').hide();
    });
});
```

More jquery Selectors :

Syntax	Description
<code>\$("*")</code>	Selects all elements
<code>\$(this)</code>	Selects the current HTML element
<code>\$("p.intro")</code>	Selects all <code><p></code> elements with class="intro"
<code>\$("p:first")</code>	Selects the first <code><p></code> element
<code>\$("ul li:first")</code>	Selects the first <code></code> element of the first <code></code>
<code>\$("ul li:first-child")</code>	Selects the first <code></code> element of every <code></code>
<code>\$("[href]")</code>	Selects all elements with an href attribute
<code>\$("a[target='_blank']")</code>	Selects all <code><a></code> elements with a target attribute value equal to "_blank"
<code>\$("tr:even")</code>	Selects all even <code><tr></code> elements
<code>\$("tr:odd")</code>	Selects all odd <code><tr></code> elements

jquery Effects :

There are 3 types of jQuery Effects and they are:

- jQuery hide()
- jQuery show()
- jQuery toggle()

Syntax:

`$(selector).hide(speed, callback);`

`$(selector)show(speed, callback);`

`$(selector).toggle(speed, callback);`

Speed and callback are optional parameters

jQueryEvent Methods:

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

jQuery Terminology:

- The **jQuery function** refers to the global jQuery object or the \$ function depending on the context .
- A **jQuery object** the object returned by the jQuery function that often represents a group of elements.
- **Selected elements** refers to the DOM elements that you have selected for, most likely by some CSS selector passed to the jQuery function and possibly later filtered further.

jQuery Methods**1. DOM Manipulation**

- before(), after(), append(), appendTo()

Example: Move all paragraphs in div with id “contents”

```

$("p").appendTo("#contents");
$("h1").append(" Dom Manipulation");
<body>
<h1>jQuery Dom Manipulation</h1>
<div id="contents">
    <p>jQuery is good</p>
    <p>jQuery is better</p>
    <p>jQuery is the best</p>
</div> </body>

```

2. Attributes

- `css()`, `addClass()`, `attr()`, `html()`, `val()`

Example : Setting

```
$("#img.logo").attr("align", "left");
$("#p.copyright").html("&copy; 2009 ajaxray");
$("#input#name").val("Spiderman");
```

Example : Getting

```
var allignment = $("#img.logo").attr("align");
var copyright = $("#p.copyright").html();
var username = $("#input#name").val();
```

3. Events

- `click()`, `bind()`, `unbind()`, `live()`

Example: Binding all interactions on events.

```
$(document).ready(function(){
    $("#message").click(function(){
        $(this).hide(); })
});
<span id="message" onclick="..."> blah blah </span>
```

4. Effects

- `hide()`, `fadeOut()`, `toggle()`, `animate()`

Example :When “show-cart” link clicked, slide up/down “cart” div.

```
$("#a#show-cart").click(function(){
    $("#cart").slideToggle("slow"); })
```

5. Ajax

- `load()`, `get()`, `ajax()`, `getJSON()`

Examples: Load a page in a container

```
$("#comments").load ("/get_comments.php");
$("#comments").load ("/get_comments.php", {max: 5});
```

jQuery Method Chaining

Chaining Methods, also known as Cascading, refers to repeatedly calling one method after another on an object, in one continuous line of code. This technique abounds in jQuery and other JavaScript libraries and it is even common in some JavaScript native methods.

Example:

```
$("#wrapper").fadeOut().html("Welcome, Sir").fadeIn();
```

or this:

```
str.replace("k", "R").toUpperCase().substr(0,4);
```

is not just pleasurable and convenient but also succinct and intelligible. It allows us to read code like a sentence, flowing gracefully across the page. It also frees us from the monotonous, blocky structures we usually construct.

jQuery chaining allows you to execute multiple methods in a single statement. By doing that, it removes the need for repeatedly finding the same element to execute code. It also makes the code more compact and readable.

To perform jQuery method chaining, you should append actions to one another. Usually each statement is run as a separate operation. Chaining in jQuery is used to link multiple statements together. A chained jQuery statement is executed as one operation. Therefore, it runs faster.

Unit 2: HTML5, JQuery and Ajax

jQuery

jQuery is a JavaScript Library. jQuery simplifies JavaScript programming. jQuery is a lightweight, "write less, do more", and JavaScript library. The purpose of jQuery is to make it much easier to use JavaScript on website. jQuery was originally released in January 2006 at BarCampNYC by John Resig.

jQuery is a JavaScript library that allows web developers to add extra functionality to their websites. It is open source and provided for free under the MIT license. In recent years, jQuery has become the most popular JavaScript library used in web development.

With jQuery you select (query) HTML elements and perform "actions" on them. Basic syntax is:

`$(selector).action ()`

- A \$ sign to define/ access jQuery.
- “\$()” access an element in current html document.

A (selector) to "query (or find)" HTML elements.

A jQuery action () to be performed on the element(s).

Examples:

- `$("p").hide()` - hides all elements.
- `$("#test").hide()` - hides the element with id="test".
- `$(".test").hide()` - hides all elements with class="test".

JavaScript vs. jQuery

Let us try to understand the difference between JavaScript and jquery.

Example 1 - Hide an element with id "textbox"

//JavaScript

```
document.getElementById(textbox).style.display = "none";
```

//jQuery

```
$("#textbox").hide();
```

Example 2 - Create a <h1> tag with "my text"

//JavaScript

```
var h1 = document.CreateElement("h1");
```

```
h1.innerHTML = "my text";
```

```
document.getElementsByTagName(body)[0].appendChild(h1);
```

//jQuery

```
$(body).append( $("<h1/>").html("my text") );
```

The Document Ready Event:

```
$(document).ready(function(){
```

```
    // jQuery methods go here...
```

```
})
```

This is to prevent any jQuery code from running before the document is finished loading (is ready).

- Trying to hide an element that is not created yet.
- Trying to get the size of an image that is not loaded yet

Alternate Syntax :

```
$(function){  
  
    // jQuery methods go here...  
  
    }
```

jquery Selectors:

jQuery selectors allow you to select and manipulate HTML. With jQuery selectors you can find elements based on their id, classes, types, attributes, values of attributes and much more. It's based on the existing CSS Selectors and in addition, it has some own custom selectors. All type of selectors in jQuery, start with the dollar sign and parentheses: \$().

Types of jquery selectors .

- Element selector Id
- (#) selector Class
- (.) selector

Element Selector:

The jQuery element selector selects elements based on their tag names.

Example:

```
$(document).ready(function()  
{  
    $("#button").click(function()  
    {  
        $("#p").hide();  
    });  
});
```

Id (#) Selector :

The jQuery #id selector uses the id attribute of an HTML tag to find the specific element.

Example :

```
$(document).ready(function()
{
    $('#button').click(function()
    {
        $('#test').hide();
    });
});
```

Class (.) Selector:

The jQuery class selector finds elements with a specific class.

Example :

```
$(document).ready(function()
{
    $('#button').click(function()
    {
        $('.test').hide();
    });
});
```

More jquery Selectors :

Syntax	Description
<code>\$("*")</code>	Selects all elements
<code>\$(this)</code>	Selects the current HTML element
<code>\$("p.intro")</code>	Selects all <code><p></code> elements with <code>class="intro"</code>
<code>\$("p:first")</code>	Selects the first <code><p></code> element
<code>\$("ul li:first")</code>	Selects the first <code></code> element of the first <code></code>
<code>\$("ul li:first-child")</code>	Selects the first <code></code> element of every <code></code>
<code>\$("[href]")</code>	Selects all elements with an href attribute
<code>\$("a[target='_blank']")</code>	Selects all <code><a></code> elements with a target attribute value equal to <code>"_blank"</code>
<code>\$("tr:even")</code>	Selects all even <code><tr></code> elements
<code>\$("tr:odd")</code>	Selects all odd <code><tr></code> elements

jquery Effects :

There are 3 types of jQuery Effects and they are:

- jQuery `hide()`
- jQuery `show()`
- jQuery `toggle()`

Syntax:

`$(selector).hide(speed, callback);`

`$(selector)show(speed, callback);`

`$(selector).toggle(speed, callback);`

Speed and callback are optional parameters

jQueryEvent Methods:

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

jQuery Terminology:

- The **jQuery function** refers to the global jQuery object or the \$ function depending on the context .
- A **jQuery object** the object returned by the jQuery function that often represents a group of elements.
- **Selected elements** refers to the DOM elements that you have selected for, most likely by some CSS selector passed to the jQuery function and possibly later filtered further.

jQuery Methods**1. DOM Manipulation**

- before(), after(), append(), appendTo()

Example: Move all paragraphs in div with id “contents”

```

$("p").appendTo("#contents");
$("h1").append(" Dom Manipulation");
<body>
<h1>jQuery Dom Manipulation</h1>
<div id="contents">
    <p>jQuery is good</p>
    <p>jQuery is better</p>
    <p>jQuery is the best</p>
</div> </body>

```

2. Attributes

- `css()`, `addClass()`, `attr()`, `html()`, `val()`

Example : Setting

```
$("#img.logo").attr("align", "left");
$("#p.copyright").html("&copy; 2009 ajaxray");
$("#input#name").val("Spiderman");
```

Example : Getting

```
var allignment = $("#img.logo").attr("align");
var copyright = $("#p.copyright").html();
var username = $("#input#name").val();
```

3. Events

- `click()`, `bind()`, `unbind()`, `live()`

Example: Binding all interactions on events.

```
$(document).ready(function(){
    $("#message").click(function(){
        $(this).hide(); })
});
<span id="message" onclick="..."> blah blah </span>
```

4. Effects

- `hide()`, `fadeOut()`, `toggle()`, `animate()`

Example :When “show-cart” link clicked, slide up/down “cart” div.

```
$("#a#show-cart").click(function(){
    $("#cart").slideToggle("slow"); })
```

5. Ajax

- `load()`, `get()`, `ajax()`, `getJSON()`

Examples: Load a page in a container

```
$("#comments").load ("/get_comments.php");
$("#comments").load ("/get_comments.php", {max: 5});
```

jQuery Method Chaining

Chaining Methods, also known as Cascading, refers to repeatedly calling one method after another on an object, in one continuous line of code. This technique abounds in jQuery and other JavaScript libraries and it is even common in some JavaScript native methods.

Example:

```
$("#wrapper").fadeOut().html("Welcome, Sir").fadeIn();
```

or this:

```
str.replace("k", "R").toUpperCase().substr(0,4);
```

is not just pleasurable and convenient but also succinct and intelligible. It allows us to read code like a sentence, flowing gracefully across the page. It also frees us from the monotonous, blocky structures we usually construct.

jQuery chaining allows you to execute multiple methods in a single statement. By doing that, it removes the need for repeatedly finding the same element to execute code. It also makes the code more compact and readable.

To perform jQuery method chaining, you should append actions to one another. Usually each statement is run as a separate operation. Chaining in jQuery is used to link multiple statements together. A chained jQuery statement is executed as one operation. Therefore, it runs faster.

Unit 2: HTML5, JQuery and Ajax

jquery: Callback

A callback function is executed after the current effect is 100% finished. JavaScript statements are executed line by line. However, with effects, the next line of code can be run even though the effect is not finished. This can create errors. To prevent this, you can create a callback function.

Syntax :

`$(selector).hide(speed, callback);`

Example : callback 1 .html

In JavaScript, statement lines are executed one by one. It might cause problems at times, as a certain effect might start running before the previous one finishes.

To prevent that, callback function jQuery comes in handy. It creates a queue of effects so they are run in a row.

CODE:

```
<!DOCTYPE html>
<html>
<head>
<script src="https://code.jquery.com/jquery-3.4.1.min.js"
  integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGW5F1Bw8HfCJo="
  crossorigin="anonymous"></script>
<script type="text/javascript" src="scripts.js"></script>
<link rel="stylesheet" href="styles.css">
</head>
<body>

<button>Click me</button>

<p>This is a paragraph make it dissappear</p>

</body>
</html>
```

```
$(document).ready(() => {  
    $("button").click(() => {  
        $("p").hide("slow", () => {  
            alert("Congratulations it works");  
        });  
    });  
});
```

Output:

Click me

This is a paragraph make it dissappear

Once you click on Click ME button the texts disappears and alert box pops up:

An embedded page on this page says
Congratulations it works

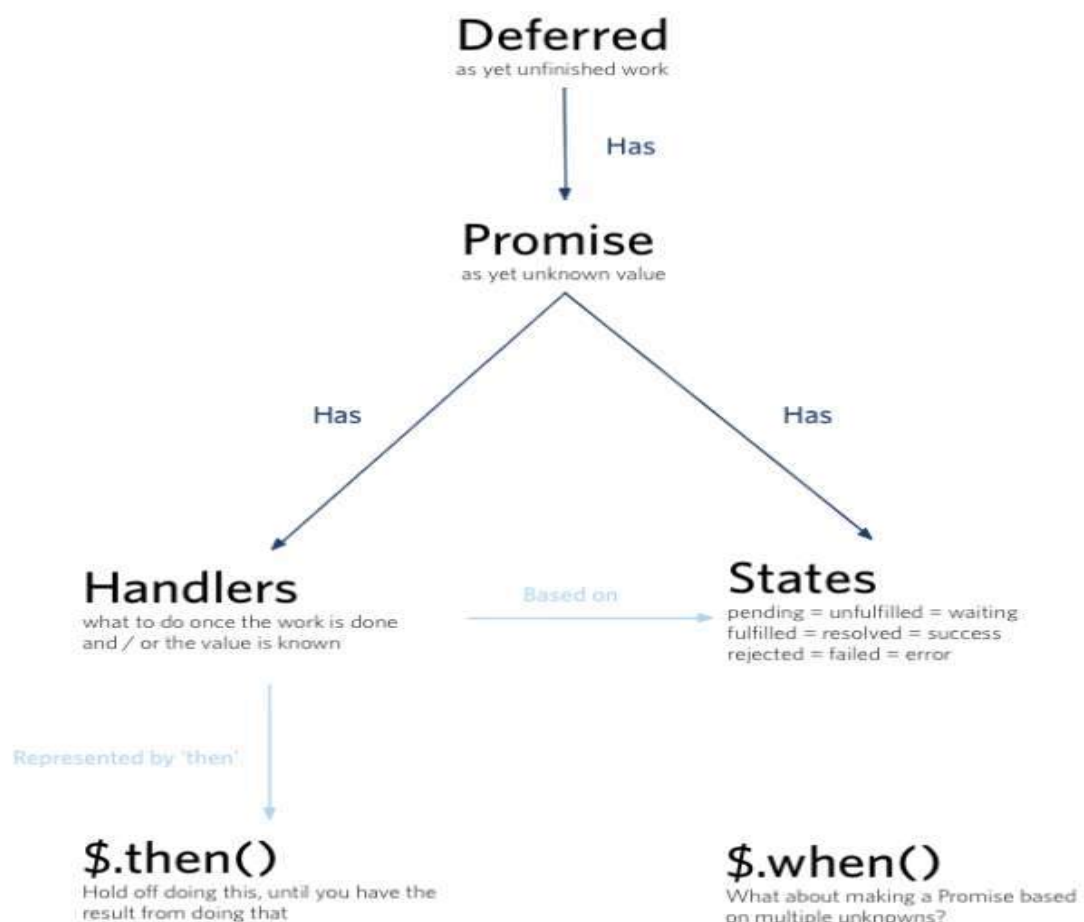
OK

Callback functions are simple and get the job done, but they become unmanageable as soon as you need to execute many asynchronous operations, either in parallel or in sequence. The situation where you have a lot of nested callbacks, or independent callbacks that have to be synchronized, is often referred to as the “callback hell.”

jQuery helps you avoid all these browser issues but, depending on the version of the library you’re using, it might do it in a slightly different manner. jQuery provides you with two objects, Deferred and Promise, that you can reliably use

in projects Lets c the concept of promises through the use of two objects: **Deferred** and **Promise**.

- A promise represents a value that is not yet known
- A deferred represents work that is not yet finished



In JavaScript, a promise is a request to resource from another website, the result of a long calculation either from your server or from a JavaScript function, or the response of a REST service (these are examples of promises), and you perform other tasks while you await the result. When the latter becomes available (the promise is resolved/fulfilled) or the request has failed (the

promise is failed/rejected), you act accordingly. Promises have been widely discussed and such discussions have resulted in two proposals:

- Promises/A
- Promises/A+.

A promise represents the eventual result of an asynchronous operation. The primary way of interacting with a promise is through its `then` method, which registers callbacks to receive either a promise's eventual value or the reason why the promise cannot be fulfilled.

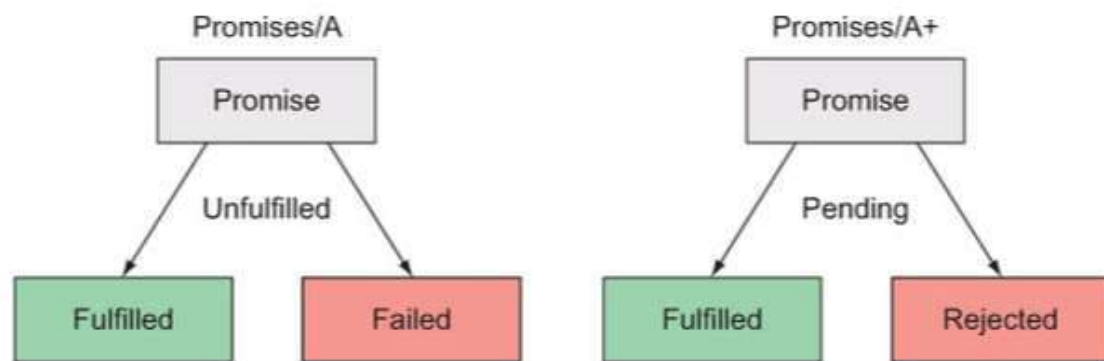
Promises/A+ specifications

The `then()` method described by the Promises/A+ proposal is the core of promises. The `then()` method accepts two functions: one to execute in the event that the promise is fulfilled and the other if the promise is rejected. When a promise is in one of these states, regardless of which one, it's settled. If a promise is neither fulfilled nor rejected (for example, if you're still waiting for the response of a server calculation), it's pending.

A promise represents the eventual value returned from the single completion of an operation. A promise may be in one of the three states, unfulfilled, fulfilled, and failed. The promise may only move from unfulfilled to fulfilled, or unfulfilled to failed.

Promises/A specifications

As per below Fig, the terminology for the Promise object's definition is a bit different. The Promises/A proposal defines the unfulfilled, fulfilled, and failed states, whereas the Promises/A+ proposal uses the pending, fulfilled, and rejected states.



These proposals outline the behavior of promises and not the implementation, so libraries implementing promises have a `then()` method in common but may differ for other methods exposed.

The Deferred and Promise objects

The Deferred object was introduced in jQuery 1.5 as a chainable utility used to register multiple callbacks into callback queues, invoke callback queues, and relay the success or failure state of any synchronous or asynchronous function. This object can be used for many asynchronous operations, like Ajax requests and animations, but also with JavaScript timing functions. Together with the Promise object, it represents the jQuery implementation of promises.

The Promise object is created starting from a Deferred object or a jQuery object and possesses a subset of the methods of the Deferred object (`always()`, `done()`, `fail()`, `state()`, and `then()`). Deferred objects are typically used if you write your own function that deals with asynchronous callbacks. So, function is the producer of the value and you want to prevent users from changing the state of the Deferred.

The Deferred methods

In jQuery, a Deferred object is created by calling the `$.Deferred()` constructor. The syntax of this function is as follows.

Method syntax: \$.Deferred

\$.Deferred([beforeStart])

A constructor function that returns a chainable utility object with methods to register multiple callbacks into callback queues, invoke callback queues, and relay the success or failure state of any synchronous or asynchronous function. It accepts an optional function to execute before the constructor returns.

Parameters

beforeStart (Function) A function that's called before the constructor returns. The function accepts a Deferred object used as the context (this) of the function.

Returns

The Deferred object.

Example:

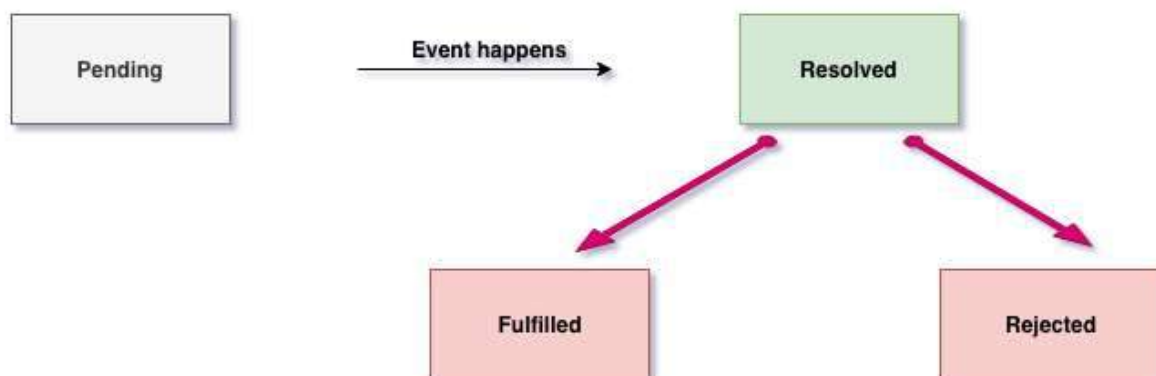
```
$.Deferred().html('Promises are great!');
```



In jQuery, a promise can be resolved (the promise is successful), rejected (an error occurred), or pending (the promise is neither resolved nor rejected). These states can be reached in two ways. The first is determined by code that you or another developer has written and with an explicit call to methods like `deferred.resolve()`, `deferred.resolveWith()`, `deferred.reject()`, or `deferred.rejectWith()`. These methods, as we'll discuss in a few moments, allow you to either resolve or reject a promise. The second is determined by the success or the failure of a jQuery function—for example, `$.ajax()`—so you don't have to call any of the previously mentioned methods yourself.

A promise is an object that represents the return value or the thrown exception that the function may eventually provide. In other words, a promise represents a value that is not yet known. A promise is an asynchronous value. The core idea behind promises is that a promise represents the result of an asynchronous operation. A Promise has 3 possible states – Pending – Fulfilled – Rejected.

THREE STATES OF A PROMISE



Example: Simple Functional Transform

```
var author = getAuthors();
```

```
var authorName = author.name;
```

// becomes

```
var authorPromise = getAuthors().then(function (author) {
```

```
  return author.name; });
```

Method syntax: promise

```
promise([type][, target])
```

Returns a dynamically generated Promise object that's resolved once all actions of a certain type bound to the collection, queued or not, have finished. By default, type is fx, which means the returned Promise is resolved when all animations of the selected elements have completed.

Parameters

type (String) The type of queue that has to be observed. The default value is fx, which represents the default queue for the effects.

target (Object) The object onto which the promise methods have to be attached.

Returns

A Promise object.

Example:

```
$('#p')
```

```
  .promise()
```

```
.then(function(value) { console.log(value); });
```

Deferred Vs Promise

The main difference between callbacks and promises is that with callbacks you tell the executing function what to do when the asynchronous task completes, whereas with promises the executing function returns a special object to you (the promise) and then you tell the promise what to do when the asynchronous task completes.

<u>Deferred</u>	<u>Promise</u>
You cannot use \$.Promise();	new \$.Deferred(); OR \$.Deferred()
A <u>deferred</u> object is an object that can create a promise and change its state to resolved or rejected. <u>Deferreds</u> are typically used if you write your own function and want to provide a promise to the calling code. You are the producer of the value.	A promise is, as the name says, a promise about a future value. You can attach callbacks to it to get that value. The promise was "given" to you and you are the receiver of the future value. You cannot modify the state of the promise. Only the code that <i>created</i> the promise can change its state.
You can call use resolve or reject etc.	The Promise exposes only the Deferred methods needed to attach additional handlers or determine the state (then, done, fail, always, pipe, progress, and state), but not ones that change the state (resolve, reject, notify, <u>resolveWith</u> , <u>rejectWith</u> , and <u>notifyWith</u>).

Unit 2: HTML5, JQuery and Ajax

Asynchronous Communication- XHR

AJAX is not a programming language, but a technique that incorporates a client-side script (i.e. a script that runs in a user's browser) that communicates with a web server. Further, its name is somewhat misleading: while an AJAX application might use XML to send data, it could also use just plain text or JSON text. But generally, it uses an XMLHttpRequest object in your browser to request data from the server and JavaScript to display the data.

XMLHttpRequest supports both synchronous and asynchronous communications. In general, however, asynchronous requests should be preferred to synchronous requests for performance reasons.

Synchronous requests block the execution of code which causes "freezing" on the screen and an unresponsive user experience.

Fetch API

The Fetch API provides an interface for fetching resources. It will seem familiar to anyone who has used XMLHttpRequest, but this API provides a more powerful and flexible feature set.

AJAX can access the server both synchronously and asynchronously:

- **Synchronously**, in which the script stops and waits for the server to send back a reply before continuing.
- **Asynchronously**, in which the script allows the page to continue to be processed and handles the reply if and when it arrives.

Components of AJAX

The AJAX cannot work independently. It is used in combination with other technologies to create interactive Web pages that are described in the following list:

JavaScript:

- Loosely typed scripting language.
- JavaScript function is called when an event occurs in a page.
- Glue for the whole AJAX operation.

DOM:

- API for accessing and manipulating structured documents.
- Represents the structure of XML and HTML documents.

CSS:

- Allows for a clear separation of the presentation style from the content and may be changed programmatically by JavaScript.

XMLHttpRequest:

- JavaScript object that performs asynchronous interaction with the server.

XMLHttpRequest Methods

- **abort()**

Cancels the current request.

- **getAllResponseHeaders()**

Returns the complete set of HTTP headers as a string.

- **getResponseHeader(headerName)**

Returns the value of the specified HTTP header.

- **open(method, URL)**
- **open(method, URL, async)**
- **open(method, URL, async, userName)**
- **open(method, URL, async, userName, password)**

Specifies the method, URL, and other optional attributes of a request.

The method parameter can have a value of "GET", "POST", or "HEAD". Other HTTP methods such as "PUT" and "DELETE" (primarily used in REST applications) may be possible.

The "async" parameter specifies whether the request should be handled asynchronously or not. "true" means that the script processing carries on after the send() method without waiting for a response, and "false" means that the script waits for a response before continuing script processing.

- **send(content)**

Sends the request.

- **setRequestHeader(label, value)**

Adds a label/value pair to the HTTP header to be sent.

XMLHttpRequest Properties

- **onreadystatechange**

An event handler for an event that fires at every state change.

- **readyState**

The readyState property defines the current state of the XMLHttpRequest object.

The following table provides a list of the possible values for the `readyState` property –

State	Description
0	The request is not initialized.
1	The request has been set up.
2	The request has been sent.
3	The request is in process.
4	The request is completed.

readyState = 0 After you have created the `XMLHttpRequest` object, but before you have called the `open()` method.

readyState = 1 After you have called the `open()` method, but before you have called `send()`.

readyState = 2 After you have called `send()`.

readyState = 3 After the browser has established a communication with the server, but before the server has completed the response.

readyState = 4 After the request has been completed, and the response data has been completely received from the server.

- **responseText**

Returns the response as a string.

- **responseXML**

Returns the response as XML. This property returns an XML document object, which can be examined and parsed using the W3C DOM node tree methods and properties.

- **status**

Returns the status as a number (e.g., 404 for "Not Found" and 200 for "OK").

- **statusText**

Returns the status as a string (e.g., "Not Found" or "OK").

How to choose between GET & POST?

Purpose of GET - to GET information , intended to be used when you are reading information to display on the page. Browsers will automatically cache the result from a GET request and if the same GET request is made again then they will display the cached result rather than rerunning the entire request. A GET call is retrieving data to display in the page and data is not expected to be changed on the server by such a call and so re-requesting the same data should be expected to obtain the same result.

POST method is intended to be used where you are updating information on the server . Results returned from server . A POST call will therefore always obtain the response from the server rather than keeping a cached copy of the prior response. If the value to be retrieved is expected to vary over time as a result of other processes updating it then add a current time parameter to what you are passing in your GET call. These criteria is not only for GET and POST for your Ajax calls but also to GET or POST when processing forms on your web page as well.

Unit 2: HTML5, JQuery and Ajax

AJAX load() Method

- The load() method loads data from a server and puts the returned data into the selected element.

Syntax: \$(selector).load(URL, data, callback);

- The required URL parameter specifies the URL you wish to load.
- The optional data parameter specifies a set of query string key/value pairs to send along with the request.
- The optional callback parameter is the name of a function to be executed after the load() method is completed. The following example loads the content of the file "test.txt" into a specific <div> element:

```
$("#div1").load("test.txt");
```

It is also possible to add a jQuery selector to the URL parameter. The following example loads the content of the element with id="p1", inside the file "test.txt", into a specific <div> element:

```
$("#div1").load("test.txt #p1");
```

The optional callback parameter specifies a callback function to run when the load() method is completed. The callback function can have different parameters:

- responseTxt - contains the resulting content if the call succeed
- statusTXT - contains the status of the call
- xhr - contains the XMLHttpRequest object

The following example displays an alert box after the load() method completes. If the load() method has succeed, it displays "External content loaded successfully!", and if it fails it displays an error message:

```
$("#button").click(function(){
$("#div1").load("demo_test.txt",function(responseTxt,statusTxt,xhr){
if(statusTxt=="success

alert("External content loaded successfully!");

if(statusTxt=="error")

alert("Error: "+xhr.status+": "+xhr.statusText); });

});
```

HTTP Request: GET vs. POST

Two commonly used methods for a request-response between a client and server.

- GET- Requests data from a specified resource
- POST - Submits data to be processed to a specified resource
- **\$.get(URL,callback);**
 - The required URL parameter specifies the URL you wish to request.
 - The optional callback parameter is the name of a function to be executed if the request succeeds.
- **\$.post(URL,data,callback);**
 - The required URL parameter specifies the URL you wish to request.

- The optional data parameter specifies some data to send along with the request.
- The optional callback parameter is the name of a function to be executed if the request succeeds.

Example

```
$("#button").click(function(){  
  
$.post("test_post.jsp",  
  
{  
  
name:"Bill Gates",  
  
city:"Seattle"  
  
},  
  
function(data,status)  
  
{  
  
alert("Data: " + data + "\nStatus: " + status);  
  
});  
  
});
```

- The first parameter of \$.post() is the URL we wish to request ("test_post.jsp").
- Then we pass in some data to send along with the request (name and city).

- The JSP script in "test_post.jsp" reads the parameters, process them, and return a result.
- The third parameter is a callback function. The first callback parameter holds the content of the page requested, and the second callback parameter holds the status of the request.

The \$.ajax(settings) or \$.ajax(url, settings)

Used for sending an Ajax request. The settings is an object of key-value pairs.

The frequently-used keys are:

- *url*: The request URL, which can be placed outside the *settings* in the latter form.
- *type*: GET or POST.
- *data*: Request parameters (name=value pairs). Can be expressed as an object (e.g., {name:"peter", msg:"hello"}), or query string (e.g., "name=peter&msg=hello").
- *dataType*: Expected response data type, such as text, xml, json, script or html.
- *headers*: an object for request header key-value pairs. The header X-Requested-With:XMLHttpRequest is always added.

Ajax request, by default, is asynchronous. In other words, once the .ajax() is issued, the script will not wait for the response, but continue into the next statement, so as not to lock up and freeze the screen.

NOTE: \$ is a shorthand (alias) for the jQuery object. \$() is an alias for jQuery() function for Selector. \$.ajax() is a global function (similar to class method in an OO language).

The Fetch API

The Fetch API provides a fetch() method defined on the window object, which you can use to perform requests. This method returns a Promise that can be used to retrieve the response of the request.

The fetch method only has one mandatory argument, which is the URL of the resource you wish to fetch. The Fetch API is a modern interface that allows you to make HTTP requests in the web browsers. The fetch() method is available in the global scope that instructs the browser to send a request to a provided URL.

Sending a Request

The fetch() has only one parameter which most of the time is the URL of the resource that you want to fetch:

```
let response = fetch(url);
```

The fetch() method returns a Promise so you can use the then() and catch() methods to handle it:

fetch(url)

```
.then(response => {  
  
    // handle the response  
  
})  
  
.catch(error => {  
  
    // handle the error  
  
});
```

When the request completes, the resource is available. At this time, the promise will resolve into a Response object.

The Response object is the API wrapper for the fetched resource. The Response object has a number of useful properties and methods to inspect the response.

Reading a Response

If the contents of the response are in the raw text format, you can use the `text()` method. The `text()` method returns a Promise that resolves with the complete contents of the fetched resource:

```
fetch('/readme.txt')  
  
  .then(response => response.text())  
  
  .then(data => console.log(data));
```

Or `async/await` can be used:

```
async function fetchText() {  
  
  let response = await fetch('/readme.txt');  
  
  let data = await response.text();  
  
  console.log(data);  
  
}
```

Besides the `text()` method, the `Response` object has other methods such as `json()`, `blob()`, `formData()` and `arrayBuffer()` to handle respective data.

Handling status codes of a response

The `Response` object provides the status code and status text via the `status` and `statusText` properties. When a request is successful, the status code is 200 and status text is OK:

```
async function fetchText() {  
  
  let response = await fetch('/readme.txt');
```

```
console.log(response.status); // 200

console.log(response.statusText); // OK

if (response.status === 200) {

    let data = await response.text();

    // handle data

}

}
```

fetchText();

Output:

200

OK

If the requested resource doesn't exist, the response code is 404:

```
let response = await fetch('/non-existence.txt');
```

```
console.log(response.status); // 400
```

```
console.log(response.statusText); // OK
```

Output:

400

Not Found

If the requested URL throws a server error, the response code will be 500.

If the requested URL is redirected to the new one with the response 300-309, the status of the Response object is set to 200. In addition the redirected property is set to true.

The fetch() returns a promise that rejects when a real failure occurs such as a web browser timeout, a loss of network connection, and a CORS violation.