

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.

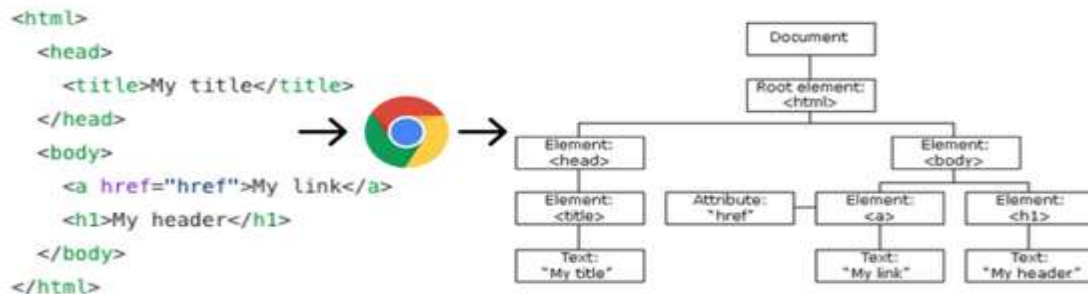
The Document Object Model (DOM) represents that same document so it can be manipulated. The DOM is an object-oriented representation of the web page, which can be modified with a scripting language such as JavaScript.

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>

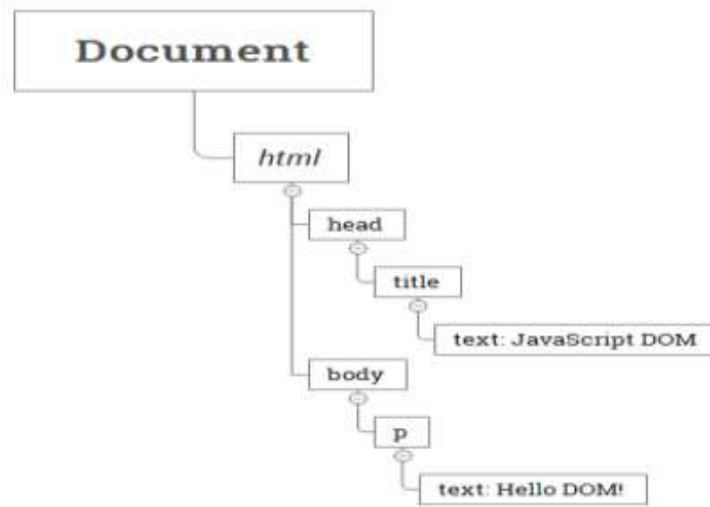
<body>

<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

A node is a generic

An element is a node with a specific node type `Node.ELEMENT_NODE`,

In other words, the node is generic type of the element. The element is a

The following picture illustrates the relationship between the Node and Element



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

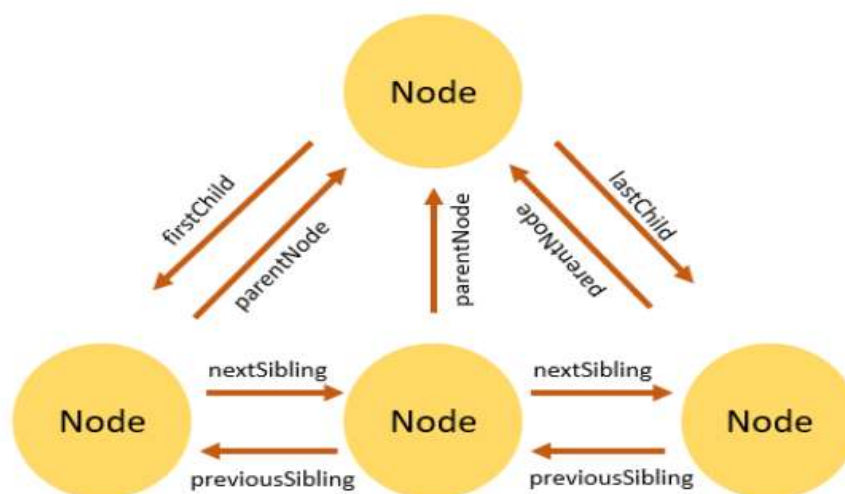
Node Relationships

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;
```

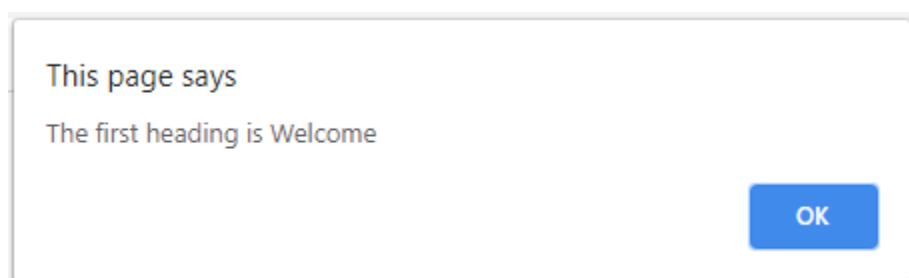
```
element.innerHTML = htmlString;
```

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>-

```

```

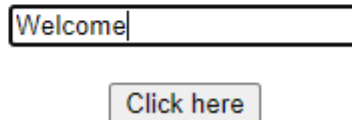
<body>-
....<h1>display screen</h1>-
....<h2>Example for getElementsByName() Method</h2>-
....<!-- This will create an input tag-->
....<input type="text" name="ga" />-
....<br>-
....<br>-
....<!-- function will be called when we
....click on this button-->
....<input type="button" onclick="display()"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

```

Results:

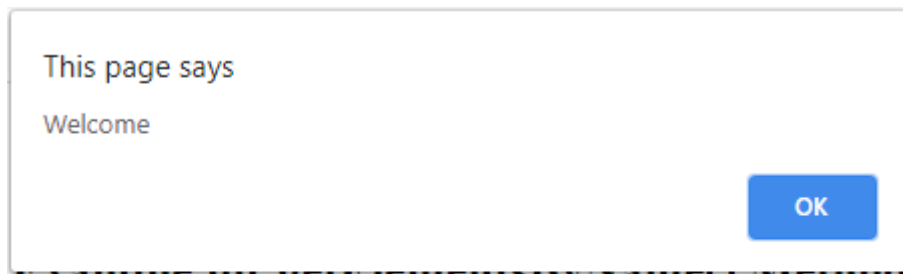
display screen

Example for getElementByName() Method

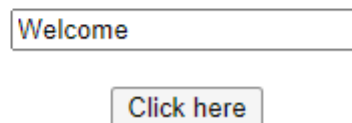


A screenshot of a web form. It features a text input field with the text 'Welcome' inside. Below the input field is a button labeled 'Click here'.

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 shows 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.