JavaScript III

Events, Forms, DOM

(Based on Tutorialpoints.com)

1. EVENTS

What is an Event?

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page.

When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc.

Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable.

Events are a part of the Document Object Model (DOM) Level 3 and every HTML element contains a set of events which can trigger JavaScript Code.

Please go through this small tutorial for a better understanding HTML Event Reference. Here we will see a few examples to understand the relation between Event and JavaScript.

onclick Event Type

This is the most frequently used event type which occurs when a user clicks the left button of his mouse. You can put your validation, warning etc., against this event type.

Example

Try the following example.

```
<html>
<head>
<script type="text/javascript">
<!--
function sayHello() {
   document.write ("Hello World")
}
//-->
</script>
```

```
</head>
<body>
<p> Click the following button and see result</p>
<input type="button" onclick="sayHello()" value="Say Hello" />
</body>
</html>
```

Output

onsubmit Event Type

onsubmit is an event that occurs when you try to submit a form. You can put your form validation against this event type.

Example

The following example shows how to use onsubmit. Here we are callinga **validate()** function before submitting a form data to the webserver. If **validate()** function returns true, the form will be submitted, otherwise it will not submit the data.

Try the following example.

```
<html>
<head>
<script type="text/javascript">
<!--
function validation() {
   all validation goes here
   .........
   return either true or false
}
//-->
</script>
</head>
```

```
<body>
<form method="POST" onsubmit="return validate()">
.......
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

onmouseover and onmouseout

These two event types will help you create nice effects with images or even with text as well. The **onmouseover** event triggers when you bring your mouse over any element and the **onmouseout** triggers when you move your mouse out from that element. Try the following example.

```
<html>
<head>
<script type="text/javascript">
<!--
function over() {
   document.write ("Mouse Over");
}
function out() {
   document.write ("Mouse Out");
}
//-->
</script>
</head>
<body>
<p>Bring your mouse inside the division to see the result:</p>
<div onmouseover="over()" onmouseout="out()">
<h2> This is inside the division </h2>
</div>
</body>
</html>
```

Output

HTML 5 Standard Events

The standard HTML 5 events are listed here for your reference. Here script indicates a Javascript function to be executed against that event.

Attribute	Value	Description
Offline	script	Triggers when the document goes offline
Onabort	script	Triggers on an abort event
onafterprint	script	Triggers after the document is printed
onbeforeonload	script	Triggers before the document loads
onbeforeprint	script	Triggers before the document is printed
onblur	script	Triggers when the window loses focus
oncanplay	script	Triggers when media can start play, but might has to stop for buffering
oncanplaythrough	script	Triggers when media can be played to the end, without stopping for buffering
onchange	script	Triggers when an element changes
onclick	script	Triggers on a mouse click
oncontextmenu	script	Triggers when a context menu is triggered
ondblclick	script	Triggers on a mouse double-click
ondrag	script	Triggers when an element is dragged
ondragend	script	Triggers at the end of a drag operation

ondragenter	script	Triggers when an element has been dragged to a valid drop target
ondragleave	script	Triggers when an element leaves a valid drop target
ondragover	script	Triggers when an element is being dragged over a valid drop target
ondragstart	script	Triggers at the start of a drag operation
ondrop	script	Triggers when dragged element is being dropped
ondurationchange	script	Triggers when the length of the media is changed
onemptied	script	Triggers when a media resource element suddenly becomes empty.
onended	script	Triggers when media has reach the end
onerror	script	Triggers when an error occur
onfocus	script	Triggers when the window gets focus
onformchange	script	Triggers when a form changes
onforminput	script	Triggers when a form gets user input
onhaschange	script	Triggers when the document has change
oninput	script	Triggers when an element gets user input
oninvalid	script	Triggers when an element is invalid
onkeydown	script	Triggers when a key is pressed
onkeypress	script	Triggers when a key is pressed and released
onkeyup	script	Triggers when a key is released

onload	script	Triggers when the document loads
onloadeddata	script	Triggers when media data is loaded
onloadedmetadata	script	Triggers when the duration and other media data of a media element is loaded
onloadstart	script	Triggers when the browser starts to load the media data
onmessage	script	Triggers when the message is triggered
onmousedown	script	Triggers when a mouse button is pressed
onmousemove	script	Triggers when the mouse pointer moves
onmouseout	script	Triggers when the mouse pointer moves out of an element
onmouseover	script	Triggers when the mouse pointer moves over an element
onmouseup	script	Triggers when a mouse button is released
onmousewheel	script	Triggers when the mouse wheel is being rotated
onoffline	script	Triggers when the document goes offline
onoine	script	Triggers when the document comes online
ononline	script	Triggers when the document comes online
onpagehide	script	Triggers when the window is hidden
onpageshow	script	Triggers when the window becomes visible
onpause	script	Triggers when media data is paused
onplay	script	Triggers when media data is going to start playing

onplaying	script	Triggers when media data has start playing
onpopstate	script	Triggers when the window's history changes
onprogress	script	Triggers when the browser is fetching the media data
onratechange	script	Triggers when the media data's playing rate has changed
onreadystatechange	script	Triggers when the ready-state changes
onredo	script	Triggers when the document performs a redo
onresize	script	Triggers when the window is resized
onscroll	script	Triggers when an element's scrollbar is being scrolled
onseeked	script	Triggers when a media element's seeking attribute is no longer true, and the seeking has ended
onseeking	script	Triggers when a media element's seeking attribute is true, and the seeking has begun
onselect	script	Triggers when an element is selected
onstalled	script	Triggers when there is an error in fetching media data
onstorage	script	Triggers when a document loads
onsubmit	script	Triggers when a form is submitted
onsuspend	script	Triggers when the browser has been fetching media data, but stopped before the entire media file was fetched
ontimeupdate	script	Triggers when media changes its playing position
onundo	script	Triggers when a document performs an undo

onunload	script	Triggers when the user leaves the document
onvolumechange	script	Triggers when media changes the volume, also when volume is set to "mute"
onwaiting	script	Triggers when media has stopped playing, but is expected to resume

2. ERRORS AND EXCEPTIONS

There are three types of errors in programming: (a) Syntax Errors, (b) Runtime Errors, and (c) Logical Errors.

Syntax Errors

Syntax errors, also called **parsing errors**, occur at compile time in traditional programming languages and at interpret time in JavaScript.

For example, the following line causes a syntax error because it is missing a closing parenthesis.

```
<script type="text/javascript">
<!--
     window.print(;
//-->
</script>
```

When a syntax error occurs in JavaScript, only the code contained within the same thread as the syntax error is affected and the rest of the code in other threads gets executed assuming nothing in them depends on the code containing the error.

Runtime Errors

Runtime errors, also called **exceptions**, occur during execution (after compilation/interpretation).

For example, the following line causes a runtime error because here the syntax is correct, but at runtime, it is trying to call a method that does not exist.

```
<script type="text/javascript">
<!--
     window.printme();
//-->
</script>
```

Exceptions also affect the thread in which they occur, allowing other JavaScript threads to continue normal execution.

Logical Errors

Logic errors can be the most difficult type of errors to track down. These errors are not the result of a syntax or runtime error. Instead, they occur when you make a mistake in the logic that drives your script and you do not get the result you expected.

You cannot catch those errors, because it depends on your business requirement what type of logic you want to put in your program.

The try...catch...finally Statement

The latest versions of JavaScript added exception handling capabilities. JavaScript implements the **try...catch...finally** construct as well as the **throw** operator to handle exceptions.

You can **catch** programmer-generated and **runtime** exceptions, but you cannot **catch** JavaScript syntax errors.

Here is the **try...catch...finally** block syntax:

```
<script type="text/javascript">
<!--
try {
    // Code to run
    [break;]
} catch ( e ) {
    // Code to run if an exception occurs
    [break;]
}[ finally {
    // Code that is always executed regardless of
    // an exception occurring
}]
//-->
</script>
```

The **try** block must be followed by either exactly one **catch** block or one **finally** block (or one of both). When an exception occurs in the **try** block, the exception is placed in **e** and the **catch** block is executed. The optional **finally** blockexecutes unconditionally after try/catch.

Example

Here is an example where we are trying to call a non-existing function which in turn is raising an exception. Let us see how it behaves without **try...catch**.

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;

   document.write ("Value of variable a is : " + a );

}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
     <input type="button" value="Click Me" onclick="myFunc();" />
</form>
<p>Error will happen and depending on your browser it will give
different result.</p>
</body>
</html>
```

Output

Now let us try to catch this exception using **try...catch** and display a user- friendly message. You can also suppress this message, if you want to hide this error from a user.

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;

   try {
      document.write ("Value of variable a is : " + a );
   } catch (e) {
      document.write ("Error: " + e.description );
   }
}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

Output

You can use a **finally** block which will always execute unconditionally after the try/catch. Here is an example.

Example

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;

   try {
      document.write ("Value of variable a is : " + a );
   }catch (e) {
      document.write ("Error: " + e.description );
   }finally {
      document.write ("Finally block will always execute!" );
   }
}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
<p>Try running after fixing the problem with method name.</p>
</body>
</html>
```

Output

The throw Statement

You can use a **throw** statement to raise your built-in exceptions or your customized exceptions. Later these exceptions can be captured and you can take an appropriate action.

Example

The following example demonstrates how to use a **throw** statement.

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;
   var b = 0;

   try{
      if (b == 0){
         throw("Divide by zero error.");
      }else{
         var c = a / b;
      }
   }catch (e) {
      document.write ("Error: " + e );
   }
}
//-->
</script>
```

```
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

Output

3. FORM VALIDATION

Form validation normally used to occur at the server, after the client hadentered all the necessary data and then pressed the Submit button. If the data entered by a client was incorrect or was simply missing, the server would haveto send all the data back to the client and request that the form be resubmitted with correct information. This was really a lengthy process which used to put a lot of burden on the server.

JavaScript provides a way to validate form's data on the client's computer before sending it to the web server. Form validation generally performs two functions.

- **Basic Validation** - First of all, the form must be checked to make sure all the mandatory fields are filled in. It would require just a loop through each field in the form and check for data.
- **Data Format Validation** - Secondly, the data that is entered must be checked for correct form and value. Your code must include appropriate logic to test correctness of data.

Example

We will take an example to understand the process of validation. Here is a simple form in html format.

```
<html>
<head>
<title>Form Validation</title>
<script type="text/javascript">
<!--
// Form validation code will come here.
//-->
</script>
</head>
<body>
 <form action="" name="myForm"onsubmit="return(validate());">
 <table cellspacing="2" cellpadding="2" border="1">
 <tr>
   <td align="right">Name</td>
```

```
  <td><input type="text" name="Name" /></td>
</tr>
<tr>
  <td align="right">EMail</td>
  <td><input type="text" name="EMail" /></td>
</tr>
<tr>
  <td align="right">Zip Code</td>
  <td><input type="text" name="Zip" /></td>
</tr>
<tr>
<td align="right">Country</td>
<td>
<select name="Country">
  <option value="-1" selected>[choose yours]</option>
  <option value="1">USA</option>
  <option value="2">UK</option>
  <option value="3">INDIA</option>
</select>
</td>
</tr>
<tr>
  <td align="right"></td>
  <td><input type="submit" value="Submit" /></td>
</tr>
</table>
</form>
</body>
</html>
```

Output

Basic Form Validation

First let us see how to do a basic form validation. In the above form, we are calling **validate()** to validate data when **onsubmit** event is occurring. The following code shows the implementation of this validate() function.

```
<script type="text/javascript">
<!--
// Form validation code will come here.
function validate()
{

   if( document.myForm.Name.value == "" )
   {
     alert( "Please provide your name!" );
     document.myForm.Name.focus() ;
     return false;
   }
   if( document.myForm.EMail.value == "" )
   {
     alert( "Please provide your Email!" );
     document.myForm.EMail.focus() ;
     return false;
```

```
   }
   if( document.myForm.Zip.value == "" ||
           isNaN( document.myForm.Zip.value ) ||
           document.myForm.Zip.value.length != 5 )
   {
     alert( "Please provide a zip in the format #####." );
     document.myForm.Zip.focus() ;
     return false;
   }
   if( document.myForm.Country.value == "-1" )
   {
     alert( "Please provide your country!" );
     return false;
   }
   return( true );
}
//-->
</script>
```

Data Format Validation

Now we will see how we can validate our entered form data before submitting it to the web server.

The following example shows how to validate an entered email address. An email address must contain at least a '@' sign and a dot (.). Also, the '@' must not be the first character of the email address, and the last dot must at least be one character after the '@' sign.

Example

Try the following code for email validation.

```
<script type="text/javascript">
<!--
function validateEmail()
{
```

```
   var emailID = document.myForm.EMail.value;
   atpos = emailID.indexOf("@");
   dotpos = emailID.lastIndexOf(".");
   if (atpos < 1 || ( dotpos - atpos < 2 ))
   {
       alert("Please enter correct email ID")
       document.myForm.EMail.focus() ;
       return false;
   }
   return( true );
}
//-->
</script>
```

4. DOM

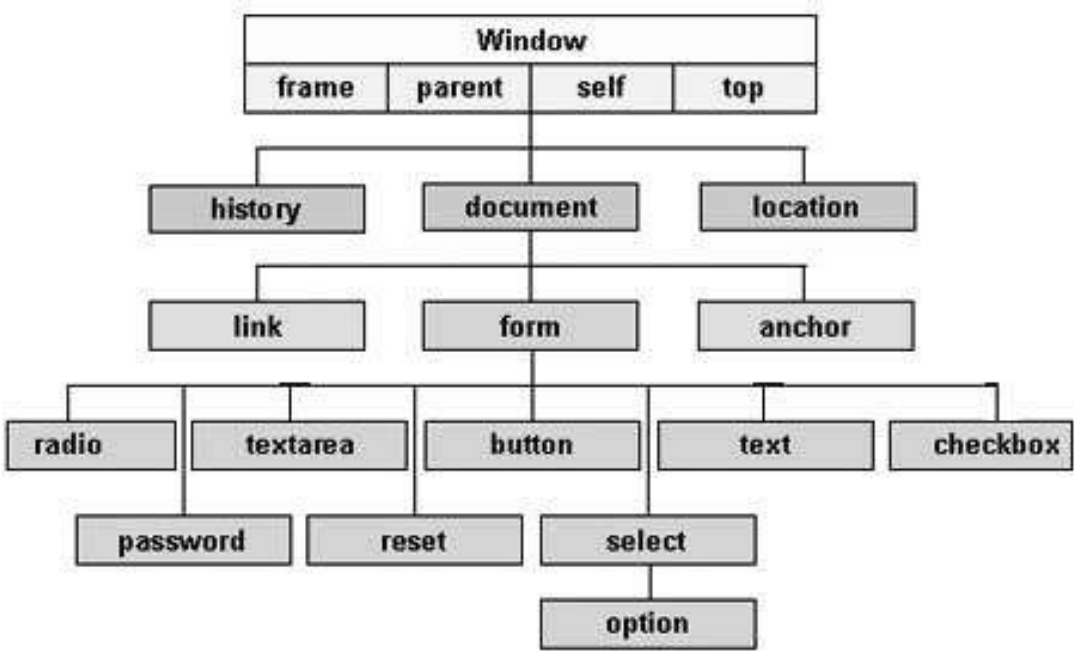
Every web page resides inside a browser window which can be considered as an object.

A Document object represents the HTML document that is displayed in that window. The Document object has various properties that refer to other objects which allow access to and modification of document content.

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.

Here is a simple hierarchy of a few important objects:



There are several DOMs in existence. The following sections explain each of these DOMs in detail and describe how you can use them to access and modify document content.

- **The Legacy DOM**: This is the model which was introduced in early versions of JavaScript language. It is well supported by all browsers, but allows accessonly to certain key portions of documents, such as forms, form elements, and images.
- **The W3C DOM**: This document object model allows access and modification of all document content and is standardized by the World Wide Web Consortium (W3C). This model is supported by almost all the modern browsers.
- **The IE4 DOM**: This document object model was introduced in Version 4 of Microsoft's Internet Explorer browser. IE 5 and later versions include support for most basic W3C DOM features.

The Legacy DOM

This is the model which was introduced in early versions of JavaScript language. It is well supported by all browsers, but allows access only to certain key portions of documents, such as forms, form elements, and images.

This model provides several read-only properties, such as title, URL, and lastModified provide information about the document as a whole. Apart from that, there are various methods provided by this model which can be used to set and get document property values.

Document Properties in Legacy DOM

Here is a list of the document properties which can be accessed using Legacy DOM.

S.No	Property and Description
1	**alinkColor** Deprecated - A string that specifies the color of activated links. **Ex:** document.alinkColor
2	**anchors[ ]** An array of Anchor objects, one for each anchor that appears in the document

	Ex: document.anchors[0], document.anchors[1] and so on
3	**applets[ ]** An array of Applet objects, one for each applet that appears in the document **Ex:** document.applets[0], document.applets[1] and so on
4	**bgColor** Deprecated - A string that specifies the background color of the document. **Ex:** document.bgColor
5	**Cookie** A string valued property with special behavior that allows the cookies associated with this document to be queried and set. **Ex:** document.cookie
6	**Domain** A string that specifies the Internet domain the document is from. Used for security purpose. **Ex:** document.domain
7	**embeds[ ]** An array of objects that represent data embedded in the document with the <embed> tag. A synonym for plugins []. Some plugins and ActiveX controls can be controlled with JavaScript code. **Ex:** document.embeds[0], document.embeds[1] and so on
8	**fgColor** A string that specifies the default text color for the document **Ex:** document.fgColor

9	**forms[ ]** An array of Form objects, one for each HTML form that appears in the document. **Ex:** document.forms[0], document.forms[1] and so on
10	**images[ ]** An array of Image objects, one for each image that is embedded in the document with the HTML <img> tag. **Ex:** document.images[0], document.images[1] and so on
11	**lastModified** A read-only string that specifies the date of the most recent change to the document **Ex:** document.lastModified
12	**linkColor** Deprecated - A string that specifies the color of unvisited links **Ex:** document.linkColor
13	**links[ ]** It is a document link array. **Ex:** document.links[0], document.links[1] and so on
14	**Location** The URL of the document. Deprecated in favor of the URL property. **Ex:** document.location
15	**plugins[ ]** A synonym for the embeds[ ]

	Ex: document.plugins[0], document.plugins[1] and so on
16	**Referrer** A read-only string that contains the URL of the document, if any, from which the current document was linked. **Ex:** document.referrer
17	**Title** The text contents of the <title> tag. **Ex:** document.title
18	**URL** A read-only string that specifies the URL of the document. **Ex:** document.URL
19	**vlinkColor** Deprecated - A string that specifies the color of visited links. **Ex:** document.vlinkColor

Document Methods in Legacy DOM

Here is a list of methods supported by Legacy DOM.

S.No	Property and Description
1	**clear( )** Deprecated - Erases the contents of the document and returns nothing. **Ex:** document.clear( )
2	**close( )** Closes a document stream opened with the open( ) method and returns nothing.

	Ex: document.close( )
3	**open( )** Deletes existing document content and opens a stream to which new document contents may be written. Returns nothing. **Ex:** document.open( )
4	**write(value, ...)** Inserts the specified string or strings into the document currently being parsed or appends to document opened with open( ). Returns nothing. **Ex:** document.write(value, ...)
5	**writeln(value, ...)** Identical to write( ), except that it appends a newline character to the output. Returns nothing. **Ex:** document.writeln(value, ...)

Example

We can locate any HTML element within any HTML document using HTML DOM. For instance, if a web document contains a **form** element, then using JavaScript, we can refer to it as **document.forms[0]**. If your Web document includes two **form** elements, the first form is referred to as document.forms[0] and the second as document.forms[1].

Using the hierarchy and properties given above, we can access the first form element using **document.forms[0].elements[0]** and so on.

Here is an example to access document properties using Legacy DOM method.

```
<html>
<head>
<title> Document Title </title>
<script type="text/javascript">
<!--
function myFunc()
```

```
{
   var ret = document.title;
   alert("Document Title : " + ret );

   var ret = document.URL;
   alert("Document URL : " + ret );

   var ret = document.forms[0];
   alert("Document First Form : " + ret );

   var ret = document.forms[0].elements[1];
   alert("Second element : " + ret );

}
//-->
</script>
</head>
<body>
<h1 id="title">This is main title</h1>
<p>Click the following to see the result:</p>

<form name="FirstForm">
<input type="button" value="Click Me" onclick="myFunc();" />
<input type="button" value="Cancel">
</form>

<form name="SecondForm">
<input type="button" value="Don't ClickMe"/>
</form>

</body>
</html>
```

NOTE: This example returns objects for forms and elements and we would have to access their values by using those object properties which are not discussed in this tutorial.

The W3C DOM

This document object model allows access and modification of all document content and is standardized by the World Wide Web Consortium (W3C). This model is supported by almost all the modern browsers.

The W3C DOM standardizes most of the features of the legacy DOM and adds new ones as well. In addition to supporting forms[ ], images[ ], and other array properties of the Document object, it defines methods that allow scripts to access and manipulate any document element and not just special-purpose elements like forms and images.

Document Properties in W3C DOM

This model supports all the properties available in Legacy DOM. Additionally, here is a list of document properties which can be accessed using W3C DOM.

S.No	Property and Description
1	**Body** A reference to the Element object that represents the <body> tag of this document. **Ex:** document.body
2	**defaultView** It is a read-only property and represents the window in which the document is displayed.

	Ex: document.defaultView
3	**documentElement** A read-only reference to the <html> tag of the document. **Ex:** document.documentElement8/31/2008
4	**Implementation** It is a read-only property and represents the DOMImplementation object that represents the implementation that created this document. **Ex:** document.implementation

Document Methods in W3C DOM

This model supports all the methods available in Legacy DOM. Additionally, here is a list of methods supported by W3C DOM.

S.No	Property and Description
1	**createAttribute(name)** Returns a newly-created Attr node with the specified name. **Ex:** document.createAttribute(name)
2	**createComment(text)** Creates and returns a new Comment node containing the specified text. **Ex:** document.createComment(text)
3	**createDocumentFragment( )** Creates and returns an empty DocumentFragment node. **Ex:** document.createDocumentFragment( )

4	**createElement(tagName)** Creates and returns a new Element node with the specified tag name. **Ex:** document.createElement(tagName)
5	**createTextNode(text)** Creates and returns a new Text node that contains the specified text. **Ex:** document.createTextNode(text)
6	**getElementById(id)** Returns the Element of this document that has the specified value for its id attribute, or null if no such Element exists in the document. **Ex:** document.getElementById(id)
7	**getElementsByName(name)** Returns an array of nodes of all elements in the document that have a specified value for their name attribute. If no such elements are found, returns a zero-length array. **Ex:** document.getElementsByName(name)
8	**getElementsByTagName(tagname)** Returns an array of all Element nodes in this document that have the specified tag name. The Element nodes appear in the returned array in the same order they appear in the document source. **Ex:** document.getElementsByTagName(tagname)
9	**importNode(importedNode, deep)** Creates and returns a copy of a node from some other document that is suitable for insertion into this document. If the deep argument is true, it recursively copies the children of the node too. Supported in DOM Version 2

	Ex: document.importNode(importedNode, deep)

Example

This is very easy to manipulate (Accessing and Setting) document element using W3C DOM. You can use any of the methods like **getElementById**, **getElementsByName**, or **getElementsByTagName**.

Here is an example to access document properties using W3C DOM method.

```
<html>
<head>
<title> Document Title </title>
<script type="text/javascript">
<!--
function myFunc()
{
   var ret = document.getElementsByTagName("title");
   alert("Document Title : " + ret[0].text );

   var ret = document.getElementById("heading");
   alert("Document URL : " + ret.innerHTML );
}
//-->
</script>
</head>
<body>
<h1 id="heading">This is main title</h1>
<p>Click the following to see the result:</p>

<form id="form1" name="FirstForm">
<input type="button" value="Click Me" onclick="myFunc();" />
<input type="button" value="Cancel">
</form>

<form d="form2" name="SecondForm">
<input type="button" value="Don't ClickMe"/>
</form>

</body>
</html>
```

NOTE: This example returns objects for forms and elements and we would have to access their values by using those object properties which are not discussed in this tutorial.

Output