



Document Object Model

MIS 4530

Dr. Garrett

Embedding JavaScript in HTML

- Put JavaScript code between opening and closing `<script>` tags. The Web browser processes an HTML page from top to bottom, executing any JavaScript it finds between `<script>` tags along the way. You can put as many `<script>` tags as you like anywhere between the opening and closing `<head>` or `<body>` tags.

```
<html>
  <body>
    <script>
      window.alert("I'm here because of JavaScript.");
    </script>
  </body>
</html>
```

- Import an external JavaScript file using the `src` attribute of the `<script>` tag. The `src` attribute specifies the URL of the file containing JavaScript code. JavaScript files typically have a `.js` extension and only contain JavaScript code. Inside the JavaScript file you do not need (and cannot have) `<script>` tags or any other HTML tags.

```
<script src="javascript/formUtil.js"></script>
```

- Assign JavaScript to the event handler attribute of an HTML tag.

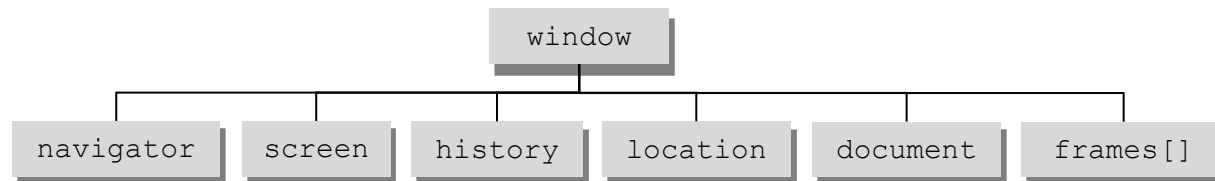
```
<button onclick="window.alert('I have been clicked!');">  
Click me!</button>
```

- Assign a single line of JavaScript code to a URL prefixed by the `javascript: pseudoprotocol`.

```
<a href="javascript: window.open('myPage.html'); void 0;">  
Click me!</a>
```

- The best practice is to import your JavaScript code using the `src` attribute of the `<script>` tag. This technique has several benefits over the others. It completely separates your JavaScript from the HTML code, which makes understanding, reading, and maintaining the code much easier. The JavaScript code can be easily reused across different Web pages. The JavaScript file can be cached by the Web browser, which makes future page loads, and loads of other pages that use the same file faster.
- The rest of the techniques for embedding JavaScript in HTML are discouraged, particularly the technique of assigning JavaScript code to an HTML event handler attribute and the use of the `javascript: pseudoprotocol`. You can accomplish everything you need with external JavaScript files.

The Browser Object Model (BOM)



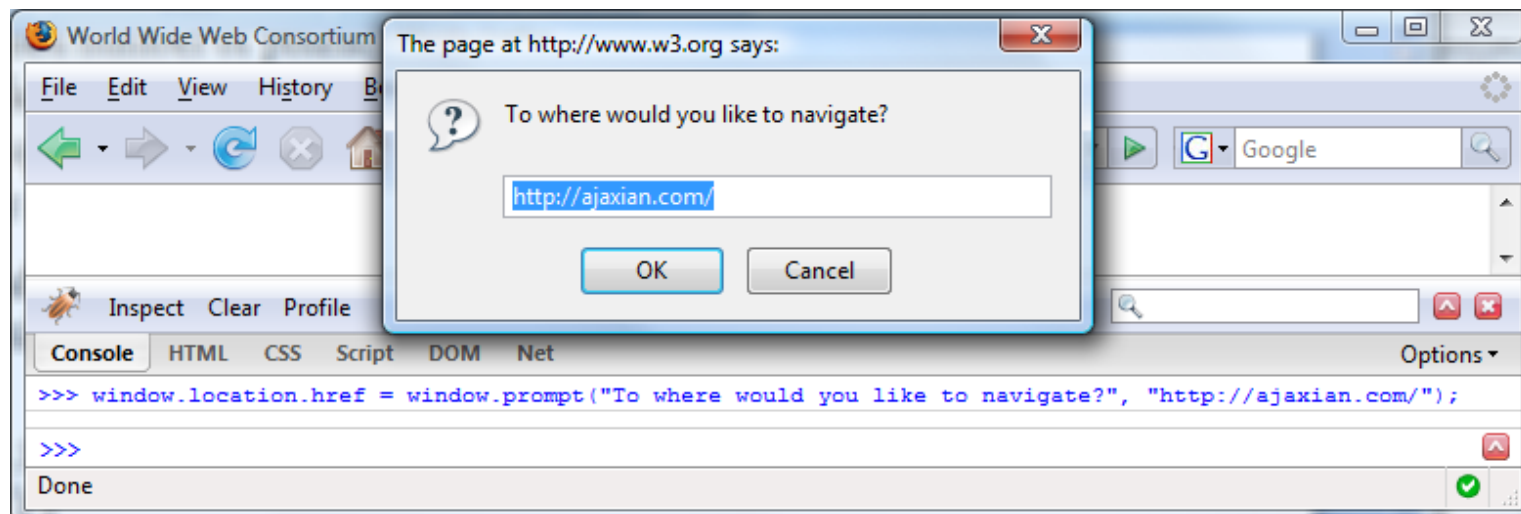
- BOM defines a set of host objects and their relationships, which are used to manipulate or obtain information about the browser
- BOM is a de-facto standard that you can rely on all modern browsers providing, however, it is not exactly the same in each browser – only a core set of objects are the same
- The `document` object contains the Document Object Model (DOM)
- The `frames[]` array contains references to all the frames loaded in the browser window

window: The global object in client-side JavaScript. Properties of the `window` object can be referenced with or without qualifying them with “`window.`”. Some properties and methods of the `window` object are given below.

- `document` – The document object (DOM).
- `frames` – An array of all the frames loaded in the window.
- `history` – The history object.
- `location` – The location object.
- `alert()` – Displays a dialog box with a message and an OK button.
- `open()` – Opens a new browser window.
- `setTimeout()` – Evaluates an expression after a specified number of milliseconds.

- The following is an example using the `window` object, and the result when executed in Firebug. Selecting the OK button will navigate the browser to the Web site `http://ajaxian.com`. Or you could enter a different address.

```
window.location.href =  
    window.prompt("To where would you like to navigate?",  
                 "http://ajaxian.com/");
```

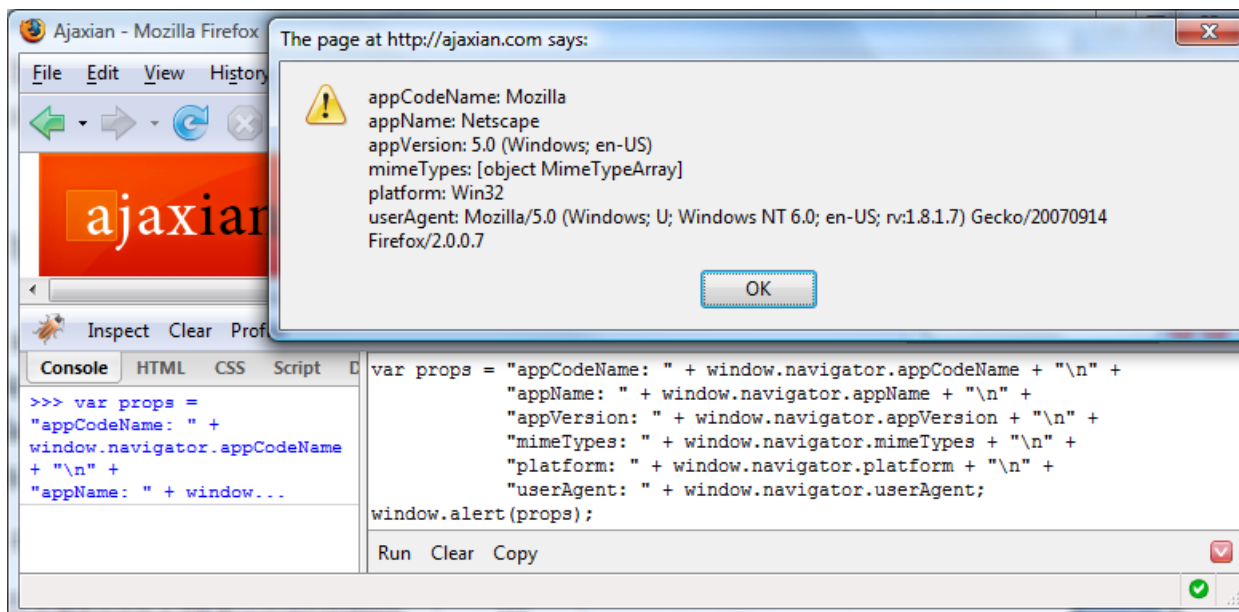


navigator: Provides information about the browser including the browser vendor, version, platform, language, and supported MIME types. The following are commonly used properties.

- `appName` – The internal code name of the browser.
- `appVersion` – The official name of the browser.
- `appVersion` – The version of the browser as a string.
- `mimeTypes` – A list of the MIME types supported by the browser.
- `platform` – The hardware platform on which the browser is running.
- `userAgent` – The string the browser sends in its USER-AGENT HTTP header.

- The following is an example that displays the value of the navigator properties listed above.

```
var props = "appCodeName: " +
    window.navigator.appCodeName + "\n" +
    "appName: " + window.navigator.appName + "\n" +
    "appVersion: " + window.navigator.appVersion + "\n" +
    "mimeTypes: " + window.navigator.mimeTypes + "\n" +
    "platform: " + window.navigator.platform + "\n" +
    "userAgent: " + window.navigator.userAgent;
window.alert(props);
```



screen: encapsulates information about the display on which the window is being rendered, including the size of the screen and the number of colors it can display. Some properties of the screen object are listed below.

- `availHeight`, `availWidth` – The height and width of the screen in pixels, minus the space required by operating system features, like a desktop taskbar.
- `height`, `width` – The height and width of the screen in pixels.
- `colorDepth` – The bit depth of the color palette of the screen.

The following example displays the width of the screen in pixels.

```
window.alert(window.screen.width());
```

history: maintains a list of the browser window's recently visited URLs. Scripts cannot directly access this list; however, scripts can call methods of the `history` object to move forward or backward in the list, similar to using the browser's Back and Forward buttons. The `history` object methods are listed below.

- `back()` – Move backward in a window's (or frame's) browsing history.
- `forward()` – Move forward in a window's (or frame's) browsing history.
- `go()` – Takes an integer argument and will move any number forward (for positive arguments) or backward (for negative arguments) in the browsing history.

```
window.history.go(-3);
```

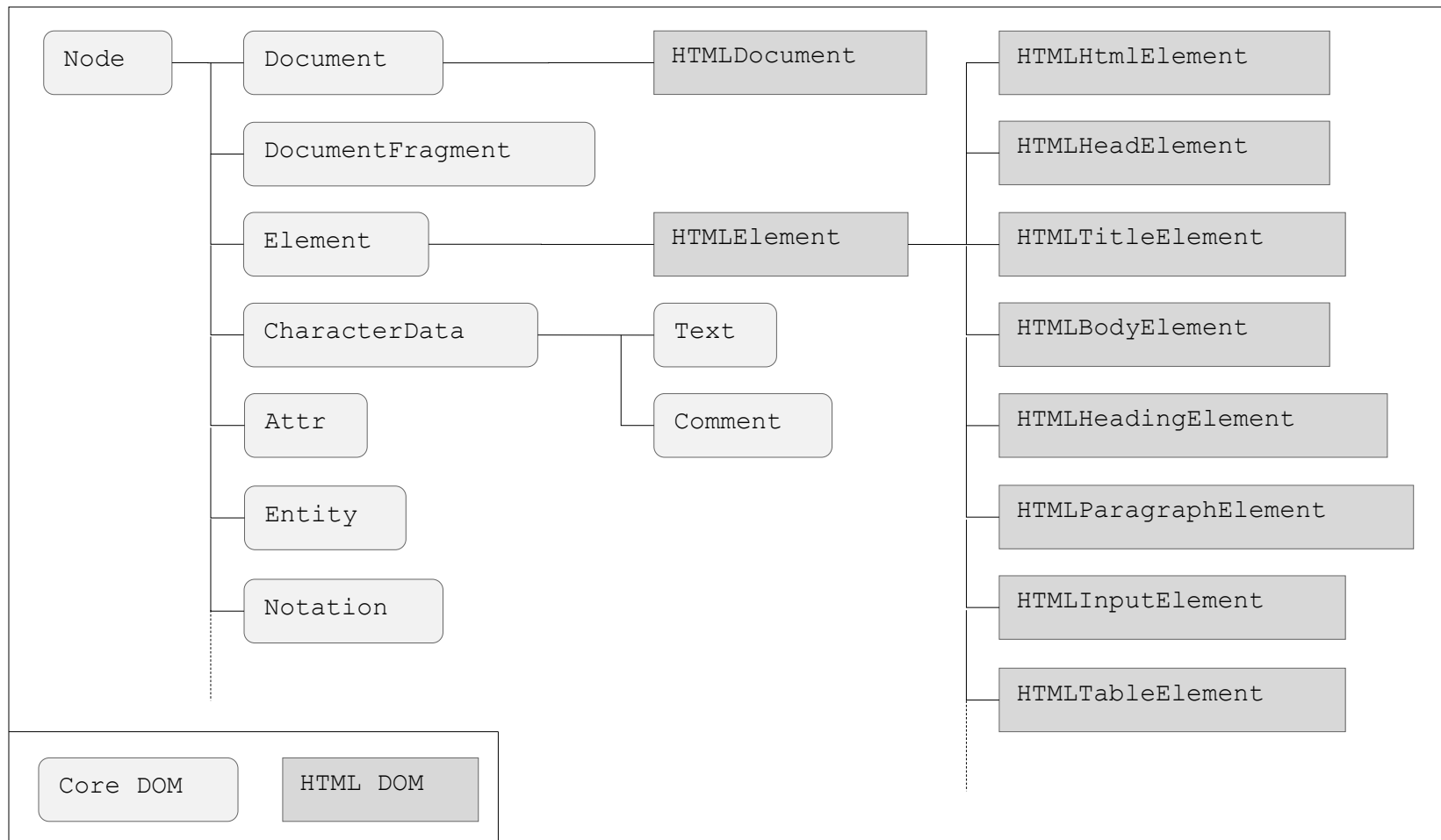
location: encapsulates information about the document currently loaded in the browser window (or frame). You can use the `location` object to change the document loaded by the browser window by assigning a URL to `window.location.href`. Some properties of the `location` object are listed below.

- `host` – The host name and port number (`www.w3.org:80`).
- `hostname` – The host name without the port number (`www.w3.org`).
- `href` – The entire URL (`http://www.w3.org:80/TR/html401/`).
- `pathname` – The path, relative to the host (`/TR/html401`).
- `port` – The port number of the URL (`80`).
- `protocol` – The protocol of the URL (`http:`).

frames []: If the page being viewed has frames (`<frameset>` or `<iframe>`), then it contains a `window` object for the page, as well as one `window` object for each frame. The `window` object for the page containing the frames has a built-in `frames []` array with references to the `window` object for each of its frames. Hence, `window.frames[0]` refers to the `window` object of the first frame. The `window` object of the top-level document (the page being viewed that containing all the frames) can be referenced from anywhere using the built-in `top` property. Because frames can be nested to any arbitrary depth there's also a built-in `parent` property that refers to the containing `window` of the current frame. In the top-level window, the properties `self`, `window`, `parent`, and `top` are all self references.

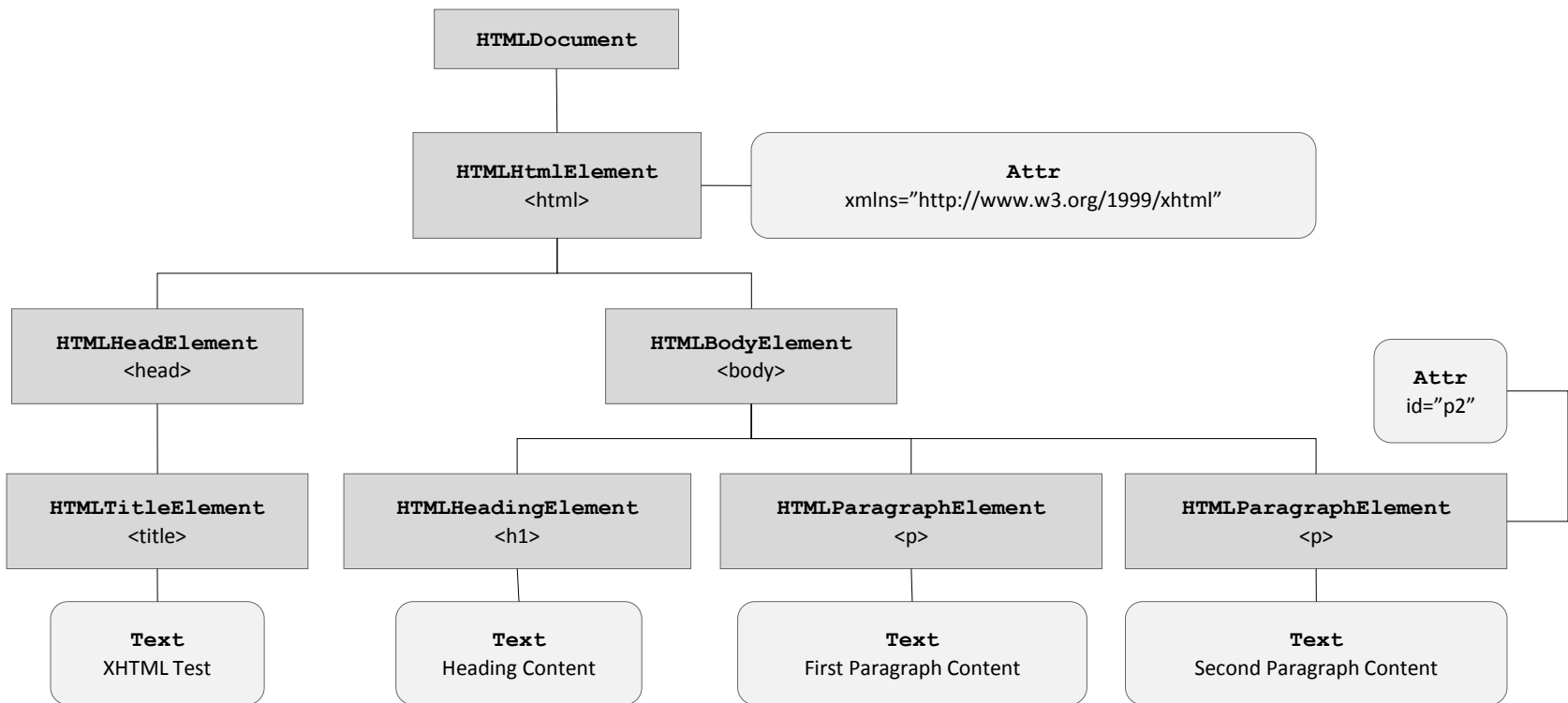
Document Object Model (DOM)

- A standard platform- and language-neutral programming interface for building, accessing, and manipulating valid HTML and well-formed XML documents.
- An interface that must be implemented in an actual programming language to be useful.
- Ultimate goal is to make it possible for programmers to write applications that work properly on all browsers and servers, and on all platforms.
- A tree-based model in which the entire document is parsed and cached in memory as a tree structure of objects called *nodes*.



- When a Web browser parses an HTML document, it creates an instance of `HTMLDocument`, which encapsulates the entire document – it becomes the root of the tree structure. The Web browser's DOM parser creates objects for every part of the document, all of which implement the `Node` interface. An HTML document is mostly parsed into three basic nodes: `Element` nodes, `Text` nodes, and `Attr` nodes. For example, the following HTML parses into the following tree.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>XHTML Test</title></head>
  <body>
    <h1>Heading Content</h1>
    <p>First Paragraph Content</p>
    <p id="p2">Second Paragraph Content</p>
  </body>
</html>
```



- The terminology for the relationships between the nodes is the same as that used for family trees. The node directly above a node in the hierarchy is the *parent* of that node (e.g., `<html>` is the parent of `<body>`). All nodes have exactly one parent, except the *root* node, which has no parent (`HTMLDocument` in the figure above is the root of the HTML hierarchy; however, `<html>` is considered the root of the document that was parsed). The nodes directly below a node are the *children* of that node (e.g., `<head>` and `<html>` are both children of `<html>`). Nodes with no children are called *leaves*. Nodes at the same level and with the same parent are *siblings* (e.g., `<head>` and `<body>` are siblings). All nodes below a node in the hierarchy are the *descendants* of that node. All nodes above a node in the hierarchy are *ancestors* of that node.

DOM Accessors

Document interface

Element documentElement: The document that was parsed, which is always `<html>` for an HTML document. There is only one document element in an instance of `Document`. The `documentElement` property of the `Document` interface is a convenient reference to the document element. Example:

```
document.documentElement.getAttribute("xmlns");
```

Element getElementById(DOMString elementId): Returns the element with an `id` attribute equal to the given ID or null if no object with that ID exists. You can assign an `id` attribute to any element in an HTML document. The ID you choose must be unique within the HTML document. Example:

```
var secondParagraph = document.getElementById("p2");
```

NodeList getElementsByTagName (DOMString tagname):

Returns a `NodeList` of all elements with the given tag name or an empty list if no tags have the given name. The method also accepts an asterisk ("`*`") as a wildcard to return all elements.

```
var paragraphs = document.getElementsByTagName("p");
for (var i = 0; i < paragraphs.length; i++) {
    alert(paragraphs.item(i).firstChild.nodeValue);
    // Or use array notation: paragraphs[i].firstChild.nodeValue
}
```

Element interface

DOMString getAttribute (DOMString name): Returns the value for the attribute with the given name.

NodeList getElementsByTagName (DOMString name): Similar to the method with the same name in the `Document` interface, except only searches below the `Element`.

Node interface

DOMString nodeName

DOMString nodeValue: Returns the name of the node and the value of the node. The name of the node is the tag name for nodes representing HTML tags, such as “HEAD” for the `<head>` tag. For other nodes, it’s a value representative of the node as defined by the DOM (e.g., “#text” for `Text` nodes). The value of the node is really only useful for nodes that contain text, like the `Text` node, because for most other nodes the value is `null`.

NodeList childNodes: Returns a `NodeList` containing all of the nodes immediate children. The `childNodes` property does not return grandchildren, i.e., a child of a child. Only `Element` nodes have children.

Node parentNode

Node firstChild

Node lastChild

Node previousSibling

Node nextSibling: parentNode returns the parent node (only Element nodes are capable of being parents, but all nodes, except for Document, have a parent). firstChild returns the first node in the NodeList returned by childNodes, while lastChild returns the last node in the list. When two nodes have the same parent they are called siblings, which means both the nodes appear in the NodeList returned by the parent's childNodes property. previousSibling returns the sibling node that comes before it in the childNodes list, while nextSibling returns the node that comes after it.

DOM Modifiers

Document interface

Element createElement(DOMString tagName): Creates an `Element` of the given type. The given `tagName` is the name of the element you wish to create. The new node must be added to an document using the `Node` methods. Example:

```
var hrElement = document.createElement("hr");
```

Text createTextNode(DOMString data): Creates a new `Text` node containing the given text. The new node must be added to an document using the `Node` methods. Example:

```
var paragraphText =  
    document.createTextNode("Third Paragraph Content");
```

Node importNode(Node importedNode, boolean deep):
Imports a node from another document to this document without modifying the other document.

`importNode()` example:

```
var otherParagraph =  
    window.parent.frames[1].getElementsByTagName("p")[0];  
var newParagraph =  
    document.importNode(otherParagraph, true);
```

Element interface

`void setAttribute(DOMString name, DOMString value):`

Adds an attribute to the `Element` or, if the attribute already exists, sets the attribute to the given value. Example:

```
document.getElementsByTagName("p")[0].setAttribute("id",  
"p1");
```

`void removeAttribute(DOMString name):` Removes the attribute with the given name. If the attribute doesn't exist, then the method has no effect.

Node interface

Node insertBefore(Node newChild, Node refChild):

Inserts the new child in the `childNodes` list before the given existing child and returns the inserted node. Example:

```
var newParagraph = document.createElement("p");
var firstParagraph = document.getElementsByTagName("p")[0];
document.body.insertBefore(newParagraph, firstParagraph);
```

Node replaceChild(Node newChild, Node oldChild):

Replaces an existing child with a new child and returns the old child node. Example:

```
var hRule = document.createElement("hr");
var firstParagraph = document.getElementsByTagName("p")[0];
document.body.replaceChild(hRule, firstParagraph);
```

Node removeChild(Node oldChild): Removes the specified child and returns it. Example:

```
var firstParagraph =  
    document.getElementsByTagName("p")[0];  
document.body.removeChild(firstParagraph);
```

Node appendChild(Node newChild): Adds the given new child to the end of the childNodes list and returns the newly added node. Example:

```
var newParagraph = document.createElement("p");  
document.body.appendChild(newParagraph);
```

Manipulating Styles

- The DOM exposes imported style sheets as instances of the `CSSStyleSheet` interface. Each style sheet object exposes the style rules, which you can inspect and modify. You access the style sheet objects via the `document.styleSheets` array.
- However, the more common method of dynamically modifying style is via the `Element` interface. The DOM exposes the styles for each `Element` via the `Element`'s `style` and `className` properties.
- The `style` property returns an instance of the `CSSStyleDeclaration` interface, which contains properties of all the style attributes supported by the browser's DOM

- For example, the following code sets the color of the document's first paragraph to red.

```
var firstParagraph = document.getElementsByTagName("p")[0];
firstParagraph.style.setProperty("color", "#FF0000", "");
```

- The common way to access and set the style properties of an `Element` is via the `CSS2Properties` interface, which contains a publicly accessible property for every style attribute defined by the DOM. In most modern browsers the object returned by the `style` property implements `CSS2Properties`. For example:

```
var firstParagraph = document.getElementsByTagName("p")[0];
firstParagraph.style.color = "#FF0000";
```

- When you need to change several styles for an `Element` at the same time it is more convenient to define the rules in a style sheet for a class selector, and then use JavaScript to set the selector assigned to the `Element`'s `className` property. For example, if the following styles were defined,

```
.normal { color: #000000; font-style: normal; }  
.styled { color: #FF0000; font-style: italic; }
```

then we could use the following code to switch the styles.

```
function toggleStyles() {  
    var p2 = document.getElementById("p2");  
    p2.className = (p2.className == "normal")  
        ? "styled" : "normal";  
}
```

Handling Events

- The Web browser generates an *event* whenever something interesting happens, and you can write JavaScript code to respond to those events by registering *event handlers*. This is called event-driven programming.
- There are two standard mechanisms for even handling – DOM Level 0 and DOM Level 2.
- DOM Level 0 event handling involves assigning event handlers to HTML objects. For example, the following assigns JavaScript code to handle the event that occurs when the user clicks on the button.

```
<button onclick="document.getElementById('p2').className =  
    (document.getElementById('p2').className == 'normal') ?  
    'styled' : 'normal';">Toggle Styles</button>
```

- A better practice than assigning JavaScript code to the HTML attribute is to assign a JavaScript function to the property of the JavaScript object. This practice fully separates the HTML code and the JavaScript code, making both of them easier to read and maintain. For example:

```
function toggleStyles() {  
    var p2 = document.getElementById("p2");  
    p2.className = (p2.className == "normal")  
        ? "styled" : "normal";  
}
```

```
document.getElementsByTagName("button")[0].onclick =  
    toggleStyles;
```

- Because the event handler properties reference JavaScript functions, you can also directly invoke an event handler in your code like the following code illustrates.

```
document.getElementsByTagName("button")[0].onclick();
```

Partial List of Event Handlers

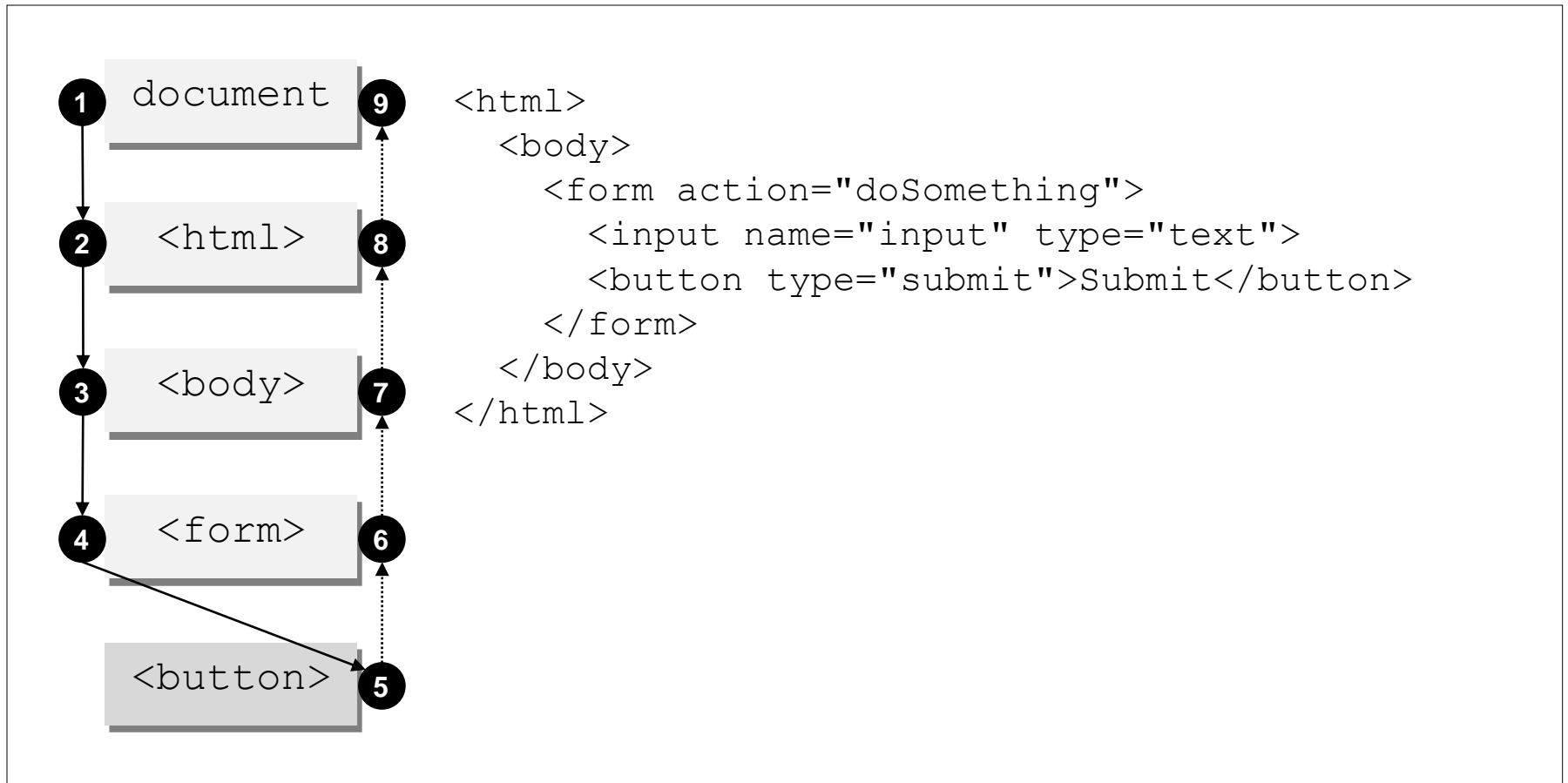
Event Handler	Trigger
<code>onblur</code>	The element loses the input focus, which is typically caused by the user clicking on another element or tabbing to another element.
<code>onchange</code>	The element loses focus <i>and</i> its value has changed since it gained focus.
<code>onclick</code>	The user clicks on the element (mouse press and release, or keyboard equivalent).
<code>onfocus</code>	The element gained the input focus.
<code>onkeypress</code>	The user pressed and released a keyboard key.
<code>onload</code>	The document finished downloading to the browser.
<code>onmousemove</code>	The user moved the mouse.
<code>onmouseout</code>	The user moved the mouse off the element.
<code>onmouseover</code>	The user moved the mouse over the element.
<code>onreset</code>	Form reset was requested; typically from the user clicking the form reset button.
<code>onselect</code>	The user selected text.
<code>onsubmit</code>	A request was made to submit a form; typically when a user clicks on the button of a form with <code>type="submit"</code> .
<code>onunload</code>	A document is about to unloaded from a window or frame.

- You can generally cancel the browser's default response to an event by returning false from the event handler. Events whose default action can be canceled by returning false from the handler are `onclick`, `onkeydown`, `onkeypress`, `onmousedown`, `onmouseup`, `onreset`, **and** `onsubmit`.

```
<script type="text/javascript">
    function validateForm(form) {
        if (form.input.value.length == 0) {
            return false;
        }
        return true;
    }
</script>
<form action="doSomething"
    onsubmit="return validateForm(this);">
    <input name="input" type="text">
    <button type="submit">Submit</button>
</form>
```

- DOM Level 0 event handling is convenient and easy to use, but it's limiting. For example, you can't register more than one handler for the same event.
- W3C defined DOM Level 2 to overcome these limitations.
- In DOM Level 2 the event is propagated through three phases:
 - (1) *Capturing phase*: the event is propagated from the top of the DOM tree down to the element on which the event occurred, called the *target node*. If any ancestors of the target node have a registered capturing event handler for the event, those handlers are run when the event passes by.
 - (2) Any event handlers registered on the *target node* are run.
 - (3) *Bubbling phase*: the event propagates back up the DOM tree (or bubbles up) and any registered event handlers for the event are executed.

The diagram below illustrates the propagation that would occur if the user clicked on the button in the listing.



- When invoked, the event handlers are passed an instance of the `Event` interface. If the event was generated by the use of the mouse, then the event is an instance of the `MouseEvent` interface, which extends the `UIEvent` interface, which in turn extends `Event`.
- During event propagation any event handler can stop further propagation by calling the `stopPropagation()` method of the `Event` object.
- If the event normally causes some default browser behavior, like following the link of an `<a>` tag, then the event handler can prevent the default behavior by calling the `preventDefault()` method of the `Event` object.

- All nodes in DOM Level 2 implement the `EventTarget` interface, which defines the methods `addEventListener()` and `removeEventListener()`, for adding/removing the event handlers to be invoked in response to a particular event.
- The first parameter specifies the event for which you want to “listen” (e.g., `submit`). The second parameter specifies the event listener function that should be invoked when the event passes by the node. The third parameter is a Boolean that specifies whether the listener is to be used during the capturing phase (`true`) or the bubbling phase (`false`).
- You can add as many listeners to a node as you like, even for the same event. They will all be invoked when the event occurs, however, the DOM does not guarantee in which order the listeners will be invoked.

- The following code demonstrates adding and removing a listener for the `submit` event on the `<submit>` tag

```
function validateForm(/* instance of Event */ eventObj) {  
    // The target is the <form>  
    var form = eventObj.target;  
    if (form.input.value.length == 0) {  
        // Prevent the form from being submitted.  
        eventObj.preventDefault();  
    }  
    // Demonstrate removing a listener.  
    form.removeEventListener("submit", validateForm, false);  
}
```

```
var form = document.getElementsByTagName("form")[0];  
// Add validateForm() as a submit event listener.  
form.addEventListener("submit", validateForm, false);
```

- All modern browsers except Microsoft Internet Explorer (IE) support the DOM Level 2 event model. The IE event model does not have a capture phase, has different event objects, and a different method for registering events.
- The IE methods for registering and unregistering event listeners are `attachEvent()` and `detachEvent()`, respectively. The IE `attachEvent()` method requires “on” to prefix the event name, and the IE event handling mechanism does not pass the event to the listener like DOM does, but instead assigns it to the `window.event` property.