DOM Enlightenment

Exploring the relationship between JavaScript and the modern HTML DOM

By Cody Lindley Version: 0.2

Introduction

This book is not an exhaustive reference on DOM scripting or JavaScript. It may, however, be the most exhaustive book written about DOM scripting without the use of a library/framework. The lack of authorship around this topic is not without good reason. Most technical authors are not willing to wrangle this topic because of the differences that exist among legacy browsers and their implementations of the DOM specifications (or lack thereof).

For the purpose of this book (i.e. grokking the concepts), I'm going to sidestep the browser API mess and dying browser discrepancies in an effort to expose the modern DOM. That's right, I'm going to sidestep the ugliness in an effort to focus on the here and now. After all, we have solutions like jQuery to deal with all that browser ugliness, and you should definitely be leveraging something like jQuery when dealing with deprecated browsers.

While I am not promoting the idea of only going native when it comes to DOM scripting, I did write this book in part so that developers may realize that DOM libraries are not always required when scripting the DOM. I also wrote for the lucky few who get to write JavaScript code for a single environment (i.e. one browser, mobile browsers, or HTML+CSS+JavaScript-to-native via something like PhoneGap). What you learn in this book may just make a DOM library unnecessary in ideal situations, say for example, some light DOM scripting for deployment on a Webkit mobile browser only.

Who should read this book

As I authored this book, I specifically had two types of developers in mind. I assume both types already have an intermediate to advanced knowledge of JavaScript, HTML, and CSS. The first developer is someone who has a good handle on JavaScript or jQuery, but has really never taken the time to understand the purpose and value of a library like jQuery (the reason for its rhyme, if you will). Equipped with the knowledge from this book, that developer should fully be able to understand the value provided by jQuery for scripting the DOM. And not just the value, but how jQuery abstracts the DOM and where and why jQuery is filling the gaps. The second type of developer is an engineer who is tasked with scripting HTML documents that will only run in modern browsers or that will get ported to native code for multiple OS's and device distributions (e.g. PhoneGap) and needs to avoid the overhead (i.e. size or size v.s. use) of a library.

Technical intentions, allowances, & limitations

- The content and code contained in this book was written with modern (IE9+, Firefox latest, Chrome latest, Safari latest, Opera latest) browsers in mind. It was my goal to only include concepts and code that are native to modern browsers. If I venture outside of this goal I will bring this fact to the readers attention. I've generally steered away from including anything in this book that is browser specific or implemented in a minority of the modern browsers.
- I'm not attempting in this book to dogmatically focus on a specific DOM, CSS, or HTML specification. Its not my goal here to dogmatically represent a specific specification. This would

be too large of an undertaking (with little value IMO) given the number of specifications at work and the history/status of browsers correctly implementing the specifications. I have leverage and balanced in a very subjective manner the content from several specifications (Document Object Model (DOM) Level 3 Core Specification, DOM4, Document Object Model HTML , Element Traversal Specification, Selectors API Level 2, DOM Parsing and Serialization, HTML 5 Reference, HTML 5 - A vocabulary and associated APIs for HTML and XHTML, HTML Living Standard, HTML 5 - A technical specification for Web Developers, DOM Living Standard). The content for this book is based more on where the community is and less on dogmatically attempting to express a specific spec.

- I'm covering several hand picked topics that are not DOM specific. I've included these topics in this book to help the reader build a proper understanding of the DOM in relationship to CSS and JavaScript.
- I've purposely left out any details as it pertains to XML or XHTML.
- I've purposely excluded the form and table api's to keep the book small. But I can see these sections being added in the future.

License

Hard Copy & eBook

O'Reilly will release and sell a hard copy & eBook in the near future.

Preface

Before you begin, it is important to understand various styles employed in this book. Please do not skip this section, because it contains important information that will aid you in the unique styles of this book.

This book is not like other programming books

The enlightenment series (jQuery Enlightenment & JavaScript Enlightenment) is written in a style that favors small, isolated, immediately executable code over wordy explanations and monolithic programs. One of my favorite authors, C.S Lewis, asserts that words are the lowest form of communication that humans traffic in. I totally agree with this assertion and use it as the basis for the style of these books. I feel that technical information is best covered with as few words as possible, in conjunction with just the right amount of executable code and commenting required to express an idea. The style of this book attempts to present a clearly defined idea with as few words as possible, backed with real code. Because of this, when you first start grokking these concepts, you should execute and examine the code, thereby forming the foundation of a mental model for the words used to describe the concepts. Additionally, the format of these books attempts to systematically break ideas down into their smallest possible form and examine each one in an isolated context. All this to say that this is not a book with lengthy explanations or in-depth coverage on broad topics. Consider yourself warned. If it helps, think of it as a cookbook, but even more terse and to the point than usual.

Color-coding conventions

In the code examples (example shown below), orange is used to highlight code directly relevant to the concept being discussed. Any additional code used to support the orange colored code will be green. The color gray in the code examples is reserved for comments.

live code: http://jsfiddle.net/domenlightenment

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

// this is a comment about a specific part of the code
var foo = 'calling out this part of the code';

</script>
</body>
</html>
```

jsFiddle

The majority of code examples in this book are linked to a corresponding jsFiddle page, where the code can be tweaked and executed online. The jsFiddle examples have been configured to use the Firebug lite-dev plugin to ensure the reader views the console.log prevalent in this book. Before reading this book, make sure you are comfortable with the usage and purpose of `console.log`.

In situations where jsFiddle caused complications with the code example, I simply chose not to link to a live example.

About the Author

Cody Lindley is a client-side engineer (aka front-end developer) and recovering Flash developer. He has an extensive background working professionally (11+ years) with HTML, CSS, JavaScript, Flash, and client-side performance techniques as it pertains to web development. If he is not wielding client-side code he is likely toying with interface/interaction design or authoring material and speaking at various conferences. When not sitting in front of a computer, it is a sure bet he is hanging out with his wife and kids in Boise, Idaho – training for triathlons, skiing, mountain biking, road biking, alpine climbing, reading, watching movies, or debating the rational evidence for a Christian worldview.

Table of Contents

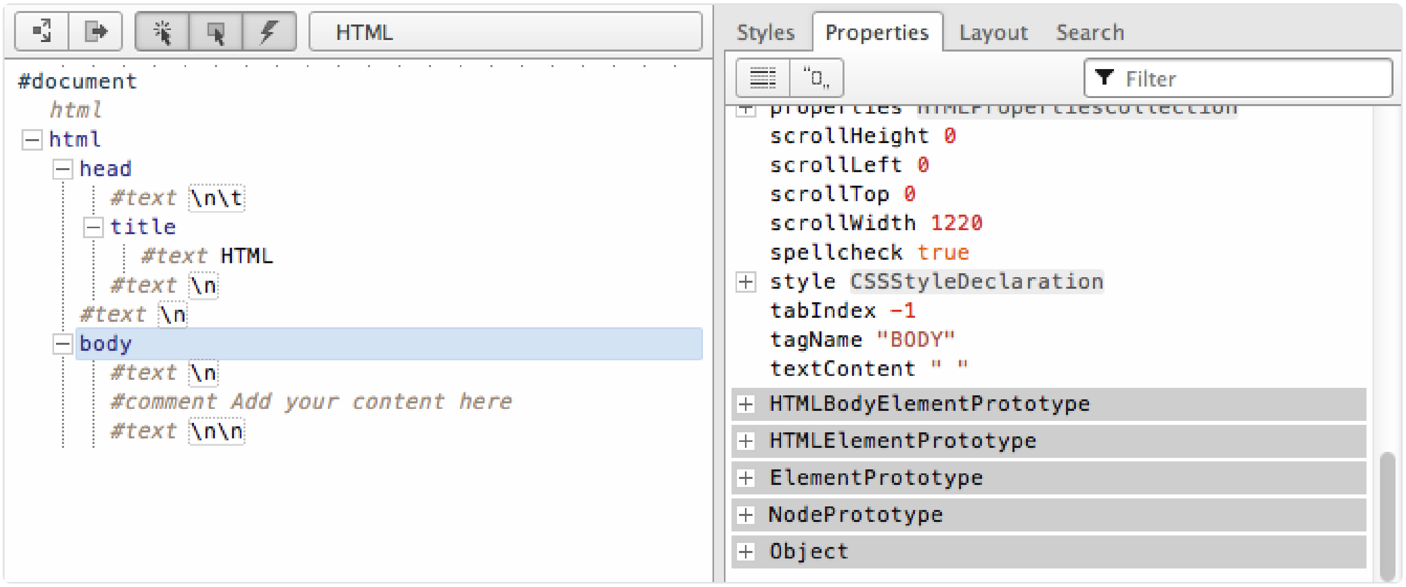
Chapter 1 - Node Overview

1.1 The Document Object Model (aka the DOM) is a hierarchy/tree of JavaScript node objects

When you write an HTML document you encapsulate HTML content inside of other HTML content. By doing this you setup a hierarchy that can be expressed as a tree. Often this hierarchy or encapsulation system is indicated visually by indenting markup in an HTML document. The browser when loading the HTML document interrupts and parses this hierarchy to create a tree of node objects that simulates how the markup is encapsulated.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>HTML</title>
</head>
<body>
<!-- Add your content here-->
</body>
</html>
```

The above HTML code when parsed by a browser creates a document that contains nodes structrured in a tree format (i.e. DOM). Below I reveal the tree struture from the above HTML document using Opera's Dragonfly DOM inspector.



```
#document
  html
html
  head
    #text \n\t
    title
      #text HTML
    #text \n
  #text \n
  body
    #text \n
    #comment Add your content here
    #text \n\n
```

```
properties HTMLPropertiesCollection
scrollHeight 0
scrollLeft 0
scrollTop 0
scrollWidth 1220
spellcheck true
style CSSStyleDeclaration
tabIndex -1
tagName "BODY"
textContent " "
HTMLBodyElementPrototype
HTMLElementPrototype
ElementPrototype
NodePrototype
Object
```

On the left you see the HTML document in its tree form. And on the right you see the corresponding JavaScript object that represents the selected element on the left. For example, the selected `<body>` element highlighted in blue, is an element node and an instance of the `HTMLBodyElement` interface.

What you should take away here is that html documents get parsed by a browser and converted into a tree structure of node objects representing a live document. The purpose of the DOM is to provide a programatic interface for scripting (removing, adding, replacing, eventing, modifiying) this live document using JavaScript.

Notes

The DOM originally was an application programming interface for XML documents that has been extended for use in HTML documents.

1.2 Node object types

The most common (I'm not highlighting all of them in the list below) types of nodes (i.e. `nodeType`/node classifications) one encounters when working with HTML documents are listed below.

- `DOCUMENT_NODE` (e.g. `window.document`)
- `ELEMENT_NODE` (e.g. `<body>`, `<a>`, `<p>`, `<script>`, `<style>`, `<html>`, `<h1>` etc...)
- `ATTRIBUTE_NODE` (e.g. `class="funEdges"`)

- `TEXT_NODE` (e.g. text characters in an html document including carriage returns and spaces)
- `DOCUMENT_FRAGMENT_NODE` (e.g. `document.createDocumentFragment()`)
- `DOCUMENT_TYPE_NODE` (e.g. `<!DOCTYPE html>`)

I've listed the node types above formatted (all uppercase with _ separating words) exactly as the constant property is written in the JavaScript browser environment as a property of the `Node` object. These `Node` properties are constant values and are used to store numeric code values which map to a specific type of node object. For example in the following code example, `Node.ELEMENT_NODE` is equal to 1. And 1 is the code value used to identify element nodes.

live code: http://jsfiddle.net/domenlightenment/BAVrs

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

console.log(Node.ELEMENT_NODE) //logs 1

</script>
</body>
</html>
```

In the code below I log all of the node types and there values.

live code: http://jsfiddle.net/domenlightenment/YcXGD

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

for(var key in Node){
    console.log(key,' = '+Node[key]);
};
/* logs to the console
ELEMENT_NODE  = 1
ATTRIBUTE_NODE  = 2
TEXT_NODE  = 3
CDATA_SECTION_NODE  = 4
ENTITY_REFERENCE_NODE  = 5
ENTITY_NODE  = 6
PROCESSING_INSTRUCTION_NODE  = 7
COMMENT_NODE  = 8
DOCUMENT_NODE  = 9
DOCUMENT_TYPE_NODE  = 10
DOCUMENT_FRAGMENT_NODE  = 11
NOTATION_NODE  = 12
DOCUMENT_POSITION_DISCONNECTED  = 1
DOCUMENT_POSITION_PRECEDING  = 2
```

```
DOCUMENT_POSITION_FOLLOWING  = 4
DOCUMENT_POSITION_CONTAINS  = 8
DOCUMENT_POSITION_CONTAINED_BY  = 16
DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC  = 32 */

</script>
</body>
</html>
```

The previous code example gives an exhaustive list of all node types. For the purpose of this book I'll be discussing the shorter list of node types listed at the start of this section. These nodes will most likely be the ones you come in contact with when scripting an HTML page.

In the table below I list the name given to the interface/constructor that instantiates the most common node types and their corresponding `nodeType` classification by number and name. What I hope you take away from the table below is the `nodeType` value (i.e. `1`) is just a numeric classificaiton used to describe a certain type of node constructed from a certain JavaScript interface/constructor. For example, the `HTMLBodyElement` interface reprsents a node object that has a node type of `1`, which is a classification for `ELEMENT_NODE`'s.

Node	
Interface:	**nodeType (returned from `.nodeType`):**
HTML*Element, (e.g. HTMLBodyElement)	**1** (i.e. `ELEMENT_NODE`)
Text	**3** (i.e. `TEXT_NODE`)
Attr	**2** (i.e. `ATTRIBUTE_NODE`)
HTMLDocument	**9** (i.e. `DOCUMENT_NODE`)
DocumentFragment	**11** (i.e. `DOCUMENT_FRAGMENT_NODE`)
DocumentType	**10** (i.e. `DOCUMENT_TYPE_NODE`)

Notes

The DOM specification semantically labels nodes like `Node`, `Element`, `Text`, `Attr`, and `HTMLAnchorElement` as an interface, which it is, but keep in mind its also the name given to the JavaScript constructor function that constructs the nodes. As you read this book I will be referring to these interfaces (i.e. `Element`, `Text`, `Attr`, `HTMLAnchorElement`) as objects or constructor functions while the specification refers to them as interfaces.

The `ATTRIBUTE_NODE` is not actually part of a tree but listed for historical reasons. In this book I do not provide a chapter on attribute nodes and instead discuss them in the `Element` node chapter given that attributes nodes are sub-like nodes of element nodes with no particiipation in the actual DOM tree structure. Be aware the ATTRIBUTE_NODE is being depreciated in DOM 4.

I've not included detail in this book on the `COMMENT_NODE` but you should be aware that comments in an HTML document are `Comment` nodes and similar in nature to `Text` nodes.

As I discuss nodes throughout the book I will rarely refer to a specific node using its `nodeType` name (e.g. `ELEMENT_NODE`). This is done to be consistent with verbiage used in the specifications provided by the W3C & WHATWG.

1.3 Sub-node objects inherit from the `Node` object

Each node object in a typical DOM tree inherits properties and methods from `Node`. Depending upon the type of node in the document there are also additional sub node object/interfaces that extend the `Node` object. Below I detail the inheritance model implemented by browsers for the most common node interfaces (< indicates inherited from).

- `Object` < `Node` < `Element` < `HTMLElement` < (e.g. `HTML*Element`)
- `Object` < `Node` < `Attr` (FYI deprecated in DOM 4)
- `Object` < `Node` < `CharacterData` < `Text`
- `Object` < `Node` < `Document` < `HTMLDocument`
- `Object` < `Node` < `DocumentFragment`

Its important not only to remember that all nodes types inherit from `Node` but that the chain of inheritance can be long. For example, all `HTMLAnchorElement` nodes inherit properties and methods from `HTMLElement`, `Element`, `Node`, and `Object` objects.

Notes

`Node` is just a JavaScript constructor function. And so logically `Node` inherits from `Object.prototype` just like all objects in JavaScript

To verify that all node types inherit properties & methods from the `Node` object lets loop over an `Element` node object and examine its properties and methods (including inherited).

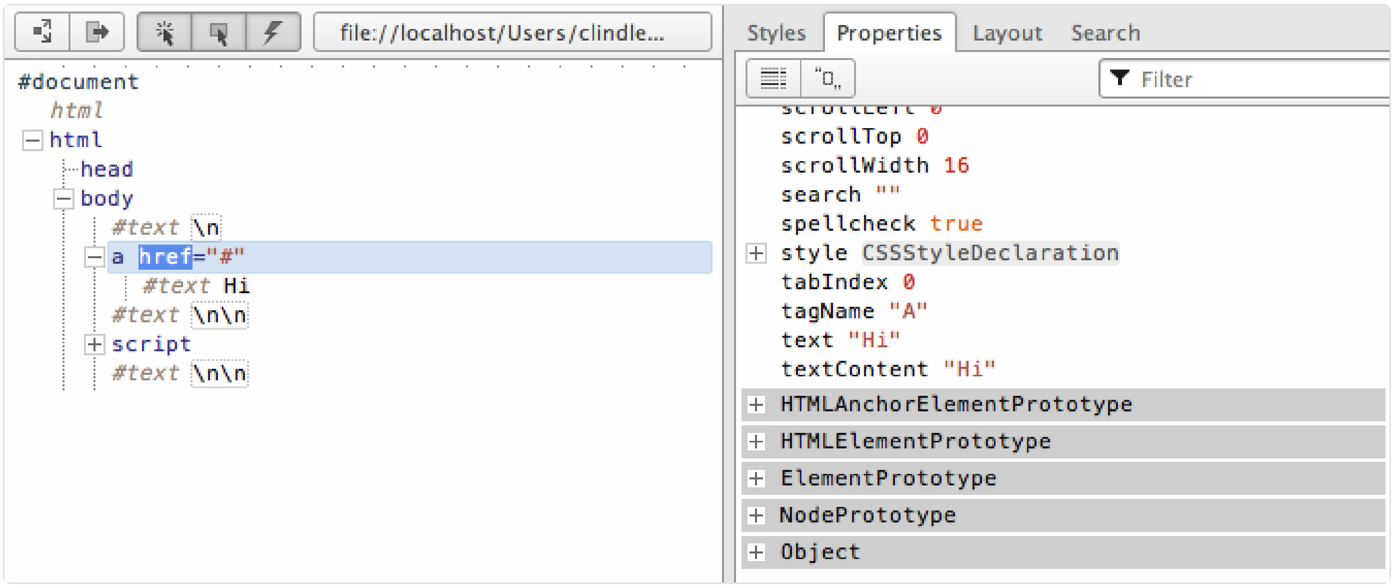
live code: http://jsfiddle.net/domenlightenment/6ukxe/

```
<!DOCTYPE html>
<html lang="en">
<body>
```

```
<a href="#">Hi</a> <!-- this is a HTMLAnchorElement which inherits from... -->

<script>

//get reference to element node object
var nodeAnchor = document.querySelector('a');

//create props array to store property keys for element node object
var props = [];

//loop over element node object getting all properties & methods (inherited too)
for(var key in nodeAnchor){
    props.push(key);
}

//log alphabetical list of properties & methods
console.log(props.sort());

</script>
</body>
</html>
```

If you run the above code in a web browser you will see a long list of properties that are available to the element node object. The properties & methods inherited from the `Node` object are in this list as well as a great deal of other inherited properties and methods from the `Element`, `HTMLElement`, `HTMLAnchorElement`, `Node`, and `Object` object. Its not my point to examine all of these properties and methods now but simply to point out that all nodes inherit a set of baseline properties and methods from its constructor as well as properties from the prototype chain.

If you are more of visual learner consider the inheritance chain denoted from examining the above HTML document with Opera's DOM inspector.



```
#document
  html
- html
    head
  - body
      #text \n
    - a href="#"
        #text Hi
      #text \n\n
    + script
      #text \n\n
```

```
  scrollTop 0
  scrollWidth 16
  search ""
  spellcheck true
+ style CSSStyleDeclaration
  tabIndex 0
  tagName "A"
  text "Hi"
  textContent "Hi"
+ HTMLAnchorElementPrototype
+ HTMLElementPrototype
+ ElementPrototype
+ NodePrototype
+ Object
```

Notice that the anchor node inherits from `HTMLAnchorElement`, `HTMLElement`, `Element`, `Node`, and `Object` all shown in the list of properties highlighted with a gray background. This inheritance chain provides a great deal of shared methods and properties to all node types.

1.4 *Node* properties and methods

Like we have been discussing all node objects (e.g `Element`, `Attr`, `Text` etc...) inherit properties and methods from a primary `Node` object. These properties and methods are the baseline values and functions for manipulating, inspecting, and traversing the DOM.

Below I list out the most common `Node` properties and methods inherited by the sub-node objects. Not all of these are discussed in this chapter. And some are not discussed at all in this book. It would be well worth your time examing a complete list of properties and methods from the DOM 4 specification or the api documention provide by mozilla. We'll be discussing and using many of them in this chapter and throughout this book.

Properties:

- `attributes`
- `childNodes`
- `firstChild`
- `lastChild`
- `nextSibling`
- `nodeName`
- `nodeType`
- `nodeValue`
- `ownerDocument`
- `parentNode`
- `previousSibling`
- `textContent`

Methods:

- `appendChild()`

- *`cloneNode()`*
- *`compareDocumentPosition()`*
- *`hasChildNodes()`*
- *`insertBefore()`*
- *`isEqualNode()`*
- *`normalize()`*
- *`removeChild()`*
- *`replaceChild()`*

1.5 Identifying the type and name of a node

Every node has a *`nodeType`* and *`nodeName`* property that is inherited from *`Node`*. For example *`Text`* nodes have a *`nodeType`* code of *`3`* and *`nodeName`* value of *`'#text'`*. As previously mentioned the numeric value *`3`* is a numeric code representing the type of underlying object the node represents (i.e. *`Node.TEXT_NODE === 3`*).

Below I detail the values returned for *`nodeType`* and *`nodeName`* for the node objects discussed in this book. It makes sense to simply memorize these numeric code's for the more common nodes given that we are only dealing with 5 numeric codes.

live code: http://jsfiddle.net/domenlightenment/8EwNu

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#">Hi</a>

<script>

//This is DOCUMENT_TYPE_NODE or nodeType 10 because Node.DOCUMENT_TYPE_NODE === 10
console.log(
        document.doctype.nodeName, //logs 'html' also try document.doctype to get <!D
OCTYPE html>
        document.doctype.nodeType //logs 10 which maps to DOCUMENT_TYPE_NODE
);

//This is DOCUMENT_NODE or nodeType 9 because Node.DOCUMENT_NODE === 9
console.log(
        document.nodeName, //logs '#document'
        document.nodeType //logs 9 which maps to DOCUMENT_NODE
);
```

```
//This is DOCUMENT_FRAGMENT_NODE or nodeType 11 because Node.DOCUMENT_FRAGMENT_NODE =
== 11
console.log(
        document.createDocumentFragment().nodeName, //logs '#document-fragment'
        document.createDocumentFragment().nodeType //logs 11 which maps to DOCUMENT_F
RAGMENT_NODE
);

//This is ELEMENT_NODE or nodeType 1 because Node. ELEMENT_NODE === 1
console.log(
        document.querySelector('a').nodeName, //logs 'A'
        document.querySelector('a').nodeType //logs 1 which maps to ELEMENT_NODE
);

//This is TEXT_NODE or nodeType 3 because Node.TEXT_NODE === 3
console.log(
        document.querySelector('a').firstChild.nodeName, //logs '#text'
        document.querySelector('a').firstChild.nodeType //logs 3 which maps to TEXT_N
ODE
);

</script>
</body>
</html>
```

If its not obvious the fastest way to determine if a node is of a certain type is too simply check its `nodeType` property. Below we check to see if the anchor element has a node number of 1. If it does than we can conclude that its an `Element` node because `Node.ELEMENT_NODE === 1`.

live code: http://jsfiddle.net/domenlightenment/ydzWL

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#">Hi</a>

<script>

//is <a> a ELEMENT_NODE?
console.log(document.querySelector('a').nodeType === 1); //logs true, <a> is an Eleme
nt node

//or use Node.ELEMENT_NODE which is a property containg the numerice value of 1
console.log(document.querySelector('a').nodeType === Node.ELEMENT_NODE); //logs true,
 <a> is an Element node

</script>
</body>
</html>
```

Determining the type of node that you might be scripting becomes very handy so that you might know which properties and methods are available to script the node.

Notes

The values returned by the *nodeName* property vary according to the node type. Have a look at the DOM 4 specification provided for the details.

1.6 Getting a nodes value

The *nodeValue* property returns *null* for most of the node types (except *Text*, and *Comment*). It's use is centered around extracting actual text strings from *Text* and *Comment* nodes. In the code below I demonstrate its use on all the nodes discussed in this book

live code: http://jsfiddle.net/domenlightenment/LNyA4

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#">Hi</a>

<script>

//logs null for DOCUMENT_TYPE_NODE, DOCUMENT_NODE, DOCUMENT_FRAGMENT_NODE, ELEMENT_NO
DE below
console.log(document.doctype.nodeValue);
console.log(document.nodeValue);
console.log(document.createDocumentFragment().nodeValue);
console.log(document.querySelector('a').nodeVale);

//logs string of text
console.log(document.querySelector('a').firstChild.nodeValue); //logs 'Hi'

</script>
</body>
</html>
```

Notes

Text or *Comment* node values can be set by providing new strings values for the *nodeValue* property(i.e. *document.body.firstElementChild.nodeValue = 'hi'*).

1.7 Creating element and text nodes using JavaScript methods

When a browser parses an HTML document it constructs the nodes and tree based on the contents of the HTML file. The browser deals with the creation of nodes for the intial loading of the HTML document. However its possible to create your own nodes using JavaScript. The following two methods allow us to programatically create `Element` and `Text` nodes using JavaScript.

- `createElement()`
- `createTextNode()`

Other methods are avaliable but are not commonly used (e.g. `createAttribute()` and `createComment()`) . In the code below I show how simple it is to create element and text nodes.

live code: http://jsfiddle.net/domenlightenment/Vj2Tc

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

var elementNode = document.createElement('div');
console.log(elementNode, elementNode.nodeType); //log <div> 1, and 1 indicates an ele
ment node

var textNode = document.createTextNode('Hi');
console.log(textNode, textNode.nodeType); //logs Text {} 3, and 3 indicates a text no
de

</script>
</body>
</html>
```

Notes

The `createElement()` method accepts one parameter which is a string specifying the element to be created. The string is the same string that is returned from the `tagName` property of an `Element` object.

The `createAttribute()` method is depricated and should not be used for creating attribute nodes. Instead developers typically use `getAttribute()`, `setAttribute()`, and `removeAttribute()` methods. I will discus this in more detail in the `Element` node chapter.

The `createDocumentFragment()` will be discussed in the chapter covering this method.

You should be aware that a `createComment()` method is available for creating comment nodes. Its not discussed in this book but is very much available to a developer who finds its usage valuable.

1.8 Creating and adding element and text nodes to the DOM using JavaScript strings

The `innerHTML`, `outerHTML`, `textContent` and `insertAdjacentHTML()` properties and methods provide the functionality to create and add nodes to the DOM using JavaScript strings.

In the code below we are using the `innerHTML`, `outerHTML`, and `textContent` properties to create nodes from JavaScript strings that are then immediately added to the DOM.

live code: http://jsfiddle.net/domenlightenment/UrNT3

```
<!DOCTYPE html>
<html lang="en">
<body>

<div id="A"></div>
<span id="B"></span>
<div id="C"></div>
<div id="D"></div>
<div id="E"></div>

<script>

//create a strong element and text node and add it to the DOM
document.getElementById('A').innerHTML = '<strong>Hi</strong>';

//create a div element and text node to replace <span id="B"></div> (notice div#B is
replaced)
document.getElementById('B').outerHTML = '<div id="B" class="new">Whats Shaking</div>
'

//create a text node and update the div#C with the text node
document.getElementById('C').textContent = 'dude';


//NON standard extensions below i.e. innerText & outerText

//create a text node and update the div#D with the text node
document.getElementById('D').innerText = 'Keep it';

//create a text node and replace the div#E with the text node (notice div#E is gone)
document.getElementById('E').outerText = 'real!';

console.log(document.body.innerHTML);
/* logs
<div id="A"><strong>Hi</strong></div>
<div id="B" class="new">Whats Shaking</div>
<span id="C">dude</span>
<div id="D">Keep it</div>
real!
*/

</script>
</body>
```

```
</html>
```

The `insertAdjacentHTML()` method which only works on `Element` nodes is a good deal more precise than the previously mentioned methods. Using this method its possible to insert nodes before the beginning tag, after the beginning tag, before the end tag, and after the end tag. Below I construct a sentence using the `insertAdjacentHTML()` method.

live code: http://jsfiddle.net/domenlightenment/tvpA6

```
<!DOCTYPE html>
<html lang="en">
<body><i id="elm">how</i>

<script>

var elm = document.getElementById('elm');

elm.insertAdjacentHTML('beforebegin', '<span>Hey-</span>');
elm.insertAdjacentHTML('afterbegin', '<span>dude-</span>');
elm.insertAdjacentHTML('beforeend', '<span>-are</span>');
elm.insertAdjacentHTML('afterend', '<span>-you?</span>');

console.log(document.body.innerHTML);
/* logs
<span>Hey-</span><i id="A"><span>dude-</span>how<span>-are</span></i><span>-you?</spa
n>
*/

</script>
</body>
</html>
```

Notes

The `innerHTML` property will convert html elements found in the string to actual DOM nodes while the `textContent` can only be used to construct text nodes. If you pass `textContent` a string containing html elements it will simply spit it out as text.

`document.write()` can also be used to simultaneously create and add nodes to the DOM. However its typically not used anymore unless its usage is required to accomplish 3rd party scripting tasks. Basically the `write()` method will output the values passed to it into the page during page loading/parsing. You should be aware that using the `write()` method will stall/block the parsing of the html document being loaded.

`innerHTML` invokes a heavy & expensive HTML parser where as text node generation is trivial thus use the innerHTML & friends sparingly

The `insertAdjacentHTML` options "beforebegin" and "afterend" will only work if the node is in the DOM tree and has a parent element.

Support for `outerHTML` was not available natively in Firefox until version 11. A polyfill is avaliable.

`textContent` gets the content of all elements, including `<script>` and `<style>` elements, `innerText` does not

`innerText` is aware of style and will not return the text of hidden elements, whereas `textContent` will

1.9 Extracting parts of the DOM tree as JavaScript strings

The same exact properties (`innerHTML`, `outerHTML`, `textContent`) that we use to create and add nodes to the DOM can also be used to extract parts of the DOM (or really the entire DOM) as a JavaScript string. In the code example below I use these properties to return a string value containing text and html values from the HTML document.

live code: http://jsfiddle.net/domenlightenment/mMYWc

```
<!DOCTYPE html>
<html lang="en">
<body>

<div id="A"><i>Hi</i></div>
<div id="B">Dude<strong> !</strong></div>

<script>

console.log(document.getElementById('A').innerHTML); //logs '<i>Hi</i>'

console.log(document.getElementById('A').outerHTML); //logs <div id="A">Hi</div>

//notice that all text is returned even if its in child element nodes (i.e. <strong>
!</strong>)
console.log(document.getElementById('B').textContent); //logs 'Dude !'

//NON standard extensions below i.e. innerText & outerText

console.log(document.getElementById('B').innerText); //logs 'Dude !'

console.log(document.getElementById('B').outerText); //logs 'Dude !'

</script>
</body>
</html>
```

Notes

The `textContent`, `innerText`, `outerText` property when being read will return all of the text nodes contained within the selected node. So for example (not a good idea in practice), `document.body.textContent` will get all the text nodes contained in the body element not just the first text node.

1.10 Adding node objects to the DOM using `appendChild()` & `insertBefore()`

The `appendChild()` and `insertBefore()` `Node` methods allow us to insert JavaScript node objects into the DOM tree. The `appendChild()` method will append a node(s) to the end of the child node(s) of the node the method is called on. If there are no child node(s) then the node being appended is appended as the first child. For example in the code below we are creating a element node (`<strong>`) and text node (`Dude`). Then the `<p>` is selected from the DOM and our `<strong>` element is appended using `appendChild()`. Notice that the `<strong>` element is encapsulated inside of the `<p>` element and added as the last child node. Next the `<strong>` element is selected and the text `'Dude'` is appended to the `<strong>` element.

live code: http://jsfiddle.net/domenlightenment/HxjFt

```
<!DOCTYPE html>
<html lang="en">
<body>

<p>Hi</p>

<script>

//create a blink element node and text node
var elementNode = document.createElement('strong');
var textNode = document.createTextNode(' Dude');

//append these nodes to the DOM
document.querySelector('p').appendChild(elementNode);
document.querySelector('strong').appendChild(textNode);

//log's <p>Hi<strong> Dude</strong></p>
console.log(document.body.innerHTML);

</script>
</body>
</html>
```

When it becomes necessary to control the location of insertion beyond appending nodes to the end of a child list of nodes we can use `insertBefore()`. In the code below I am inserting the `<li>` element before the first child node of the `<ul>` element.

live code: http://jsfiddle.net/domenlightenment/UmkME

```
<!DOCTYPE html>
<html lang="en">
<body>
```

```
<ul>
    <li>2</li>
    <li>3</li>
</ul>

<script>

//create a text node and li element node and append the text to the li
var text1 = document.createTextNode('1');
var li = document.createElement('li');
li.appendChild(text1);

//select the ul in the document
var ul = document.querySelector('ul');

/*
add the li element we created above to the DOM, notice I call on <ul> and pass refere
nce to <li>2</li> using ul.firstChild
*/
ul.insertBefore(li,ul.firstChild);

console.log(document.body.innerHTML);
/*logs
<ul>
<li>1</li>
<li>2</li>
<li>3</li>
</ul>
*/

</script>
</body>
</html>
```

The `insertBefore()` requires two parameters, the node to be inserted and the reference node in the document you would like the node inserted before.

Notes

If you do not pass the `insertBefore()` method a second parameter then it functions just like `appendChild()`.

We have more methods (e.g. `prepend()`, `append()`, `before()`, `after()`) to look forward too in DOM 4.

1.11 Removing and replacing nodes using `removeChild()` and `replaceChild()`

Removing a node from the DOM is a bit of a multi-step process. First you have to select the node you want to remove. Then you need to gain access to its parent element typically using the `parentNode` property. Its on the parent node that you invoke the `removeChild()` method passing it the reference to the node to be removed. Below I demonstrate its use on an element node and text node.

live code: http://jsfiddle.net/domenlightenment/VDZgP

```
<!DOCTYPE html>
<html lang="en">
<body>

<div id="A">Hi</div>
<div id="B">Dude</div>

<script>

//remove element node
var divA = document.getElementById('A');
divA.parentNode.removeChild(divA);

//remove text node
var divB = document.getElementById('B').firstChild;
divB.parentNode.removeChild(divB);

//log the new DOM updates, which should only show the remaining empty div#B
console.log(document.body.innerHTML);

</script>
</body>
</html>
```

Replacing an element or text node is not unlike removing one. In the code below I use the same html structure used in the previous code example except this time I use `replaceChild()` to update the nodes instead of removing them.

live code: http://jsfiddle.net/domenlightenment/zgE8M

```
<!DOCTYPE html>
<html lang="en">
<body>

<div id="A">Hi</div>
<div id="B">Dude</div>

<script>

//replace element node
var divA = document.getElementById('A');
var newSpan = document.createElement('span');
newSpan.textContent = 'Howdy';
divA.parentNode.replaceChild(newSpan,divA);

//replace text node
var divB = document.getElementById('B').firstChild;
var newText = document.createTextNode('buddy');
divB.parentNode.replaceChild(newText, divB);
```

```
//log the new DOM updates,
console.log(document.body.innerHTML);

</script>
</body>
</html>
```

Notes

Depending upon what you are removing or replacing simply providing the `innerHTML`, `outerHTML`, and `textContent` properties with an empty string might be easier and faster.

Both `replaceChild()` and `removeChild()` return the replaced or remove node. Basically its not gone just because you replace it or remove. All this does is takes it out of the current live document. You still have a reference to it in memory.

We have more methods (e.g.`replace()`, `remove()`) to look forward too in DOM 4.

1.12 Cloning nodes using `cloneNode()`

Using the `cloneNode()` method its possible to duplicate a single node or a node and all its children nodes.

In the code below I clone only the `<ul>` (i.e. `HTMLUListElement)` which once cloned can be treated like any node reference.

live code: http://jsfiddle.net/domenlightenment/6DHgC

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
  <li>Hi</li>
  <li>there</li>
</ul>

<script>

var cloneUL = document.querySelector('ul').cloneNode();

console.log(cloneUL.constructor); //logs HTMLUListElement()
console.log(cloneUL.innerHTML); //logs (an empty string) as only the ul was cloned

</script>
</body>
</html>
```

To clone a node and all of its children nodes you pass the `cloneNode()` method a parameter of of `true`. Below I use the `cloneNode()` method again but this time we clone all of the child nodes as well.

live code: http://jsfiddle.net/domenlightenment/EyFEC

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
  <li>Hi</li>
  <li>there</li>
</ul>

<script>

var cloneUL = document.querySelector('ul').cloneNode(true);

console.log(cloneUL.constructor); //logs HTMLUListElement()
console.log(cloneUL.innerHTML); //logs <li>Hi</li><li>there</li>

</script>
</body>
</html>
```

Notes

When cloning an `Element` node all attributes and values are also cloned. In fact, only attributes are copied! Everything else you can add (e.g. event handlers) to a DOM node is lost when cloning.

You might think that cloning a node and its children using `cloneNode(true)` would return a `NodeList` but it in fact does not.

1.13 Grokking node collections (i.e. `Nodelist` & `HTMLcollection`)

When selecting groups of nodes from a tree (cover in chaper 3) or accessing pre-defined sets of nodes, the nodes are either placed in a `NodeList` (e.g. `document.querySelectorAll('*')`) or `HTMLCollection` (e.g. `document.scripts`). These array like (i.e. not a real `Array`) object collections that have the following characteristics.

- A collection can either be live or static. Meaning that the nodes contained in the collection are either literally part of the live document or a snapshot of the live document.
- By default nodes are sorted inside of the collection by tree order. Meaning the order matches the

liner path from tree trunk to branches.

- The collections have a `length` property that reflects the number of elements in the list

1.14 Gettting a list/collection of all immediate child nodes

Using the `childNodes` property produces an array like list (i.e. NodeList) of the immediate child nodes. Below I select the `<ul>` element which I then use to create a list of all of the immediate child nodes contained inside of the `<ul>`.

live code: http://jsfiddle.net/domenlightenment/amDev

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
  <li>Hi</li>
  <li>there</li>
</ul>

<script>

var ulElementChildNodes = document.querySelector('ul').childNodes;

console.log(ulElementChildNodes); //logs an array like list of all nodes inside of th
e ul

/*Call forEach as if its a method of NodeLists so we can loop over the NodeList. Done
 because NodeLists are array like, but do not directly inherit from Array*/
Array.prototype.forEach.call(ulElementChildNodes,function(item){
   console.log(item); //logs each item in the array
});

</script>
</body>
</html>
```

Notes

The `NodeList` returned by `childNodes` only contains immediate child nodes

Be aware `childNodes` contains not only `Element` nodes but also all other node types (e.g. `Text` and `Comment` nodes)

`[].forEach` was implemented in ECMAScript 5th edtion

1.15 Convert a `NodeList` or `HTMLCollection` to JavaScript `Array`

Node lists and html collections are array like but not a true JavaScript array which inherits array methods. In the code below we programtically confirm this using `isArray()`.

live code: http://jsfiddle.net/domenlightenment/n53Xk

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#"></a>

<script>

console.log(Array.isArray(document.links)); //returns false, its an HTMLCollection no
t an Array
console.log(Array.isArray(document.querySelectorAll('a'))); //returns false, its an N
odeList not an Array

</script>
</body>
</html>
```

Notes

`Array.isArray` was implemented in ECMAScript 5th edtion or ES5

Converting a node list and html collection list to a true JavaScript array can provide a good deal of advantages. For one it gives us the ability to create a snapshot of the list that is not tied to the live DOM considering that `NodeList` and `HTMLCollection` are live lists. Secondly, converting a list to a JavaScript array gives access to the methods provided by the `Array` object (e.g. `forEach`, `pop`, `map`, `reduce` etc...).

To convert an array like list to a true JavaScript array pass the array-like list to `call()` or `apply()`, in which the `call()` or `apply()` is calling a method that returns an un-altered true JavaScript array. In the code below I use the `.slice()` method, which doesn't really slice anything I am just using it to convert the list to a JavaScript `Array` due to the fact the `slice()` returns an array.

live code: http://jsfiddle.net/domenlightenment/jHgTY

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#"></a>
```

```
<script>

console.log(Array.isArray(Array.prototype.slice.call(document.links))); //returns tru
e
console.log(Array.isArray(Array.prototype.slice.call(document.querySelectorAll('a')))
); //returns true

</script>
</body>
</html>
```

Notes

In ECMAScript 6th edtion we have `Array.from` to look forward to which converts a single argument that is an array-like object or list (eg. `arguments`, `NodeList`, `DOMTokenList` (used by `classList`), `NamedNodeMap` (used by `attributes` property)) into a `new Array()` and returns it

1.16 Traversing nodes in the DOM

From a node reference (i.e. `document.querySelector('ul')`) its possible to get a different node reference by traversing the DOM using the following properties:

- `parentNode`
- `firstChild`
- `lastChild`
- `nextSibling`
- `previousSibling`

In the code example below we examine the `Node` properties providing DOM traversal functionality.

live code: http://jsfiddle.net/domenlightenment/Hvfhv

```
<!DOCTYPE html>
<html lang="en">
<body><ul><!-- comment -->
<li id="A"></li>
<li id="B"></li>
<!-- comment -->
</ul>

<script>
```

```
//cache selection of the ul
var ul = document.querySelector('ul');

//What is the parentNode of the ul?
console.log(ul.parentNode.nodeName); //logs body

//What is the first child of the ul?
console.log(ul.firstChild.nodeName); //logs comment

//What is the last child of the ul?
console.log(ul.lastChild.nodeName); //logs text not comment, because there is a line
break

//What is the nextSibling of the first li?
console.log(ul.querySelector('#A').nextSibling.nodeName); //logs text

//What is the previousSibling of the last li?
console.log(ul.querySelector('#B').previousSibling.nodeName); //logs text

</script>
</body>
</html>
```

If you have been around the DOM much then it should be no surprise that traversing the DOM includes not just traversing element nodes but also text and comment nodes. I believe the last code example makes this clear, and this is not exactly ideal. Using the following properties we can traverse the DOM ignoring text and comment nodes.

- *firstElementChild*
- *lastElementChild*
- *nextElementChild*
- *previousElementChild*
- *children*

Notes

The `childElementCount` is not mentioned but you should be aware of its avaliablity for calculating the number of child elements a node contains.

Examine our code example again using only element traversing methods.

live code: http://jsfiddle.net/domenlightenment/Wh7nf

```
<!DOCTYPE html>
<html lang="en">
<body><ul><!-- comment -->
<li id="A"></li>
```

```
<li id="B"></li>
<!-- comment -->
</ul>

<script>

//cache selection of the ul
var ul = document.querySelector('ul');

//What is the first child of the ul?
console.log(ul.firstElementChild.nodeName); //logs li

//What is the last child of the ul?
console.log(ul.lastElementChild.nodeName); //logs li

//What is the nextSibling of the first li?
console.log(ul.querySelector('#A').nextElementSibling.nodeName); //logs li

//What is the previousSibling of the last li?
console.log(ul.querySelector('#B').previousElementSibling.nodeName); //logs li

//What are the element only child nodes of the ul?
console.log(ul.children); //HTMLCollection, all child nodes including text nodes

</script>
</body>
</html>
```

1.17 Verify a node position in the DOM tree with `contains()` & `compareDocumentPosition()`

Its possible to know if a node is contained inside of another node by using the `contains()` `Node` method. In the code below I ask if `<body>` is contained inside of `<html lang="en">`.

live code: http://jsfiddle.net/domenlightenment/ENU4w

```
<!DOCTYPE html>
<html lang="en">
<body>

<script>

// is <body> inside <html lang="en"> ?
var inside = document.querySelector('html').contains(document.querySelector('body'));

console.log(inside); //logs true

</script>
</body>
</html>
```

If you need more robust information about the position of a node in the DOM tree in regards to the nodes around it you can use the `compareDocumentPosition()` `Node` method. Basically this method gives you the ability to request information about a selected node relative to the node passed in. The information that you get back is a number that corresponds to the following information.

number code returned from `compareDocumentPosition()`:	**number code info:**
0	Elements are identical.
1	DOCUMENT_POSITION_DISCONNECTED Set when selected node and passed in node are not in the same document.
2	DOCUMENT_POSITION_PRECEDING Set when passed in node is preceding selected node.
3	DOCUMENT_POSITION_FOLLOWING Set when passed in node is following selected node.
8	DOCUMENT_POSITION_CONTAINS Set when passed in node is an ancestor of selected node.
16, 10	DOCUMENT_POSITION_CONTAINED_BY (16, 10 in hexadecimal) Set when passed in node is a descendant of selected node.

Notes

`contains()` will return `true` if the node selected and node passed in are identical.

`compareDocumentPosition()` can be rather confusing because its possible for a node to have more than one type of relationship with another node. For example when a node both contains (16) and precedes (4) the returned value from `compareDocumentPosition()` will be 20.

1.18 How to determine if two nodes are identical

According to the DOM 3 specification two nodes are equal if and only if the following conditions are satisfied:

- The two nodes are of the same type.
- The following string attributes are

equal: `nodeName`, `localName`, `namespaceURI`, `prefix`, `nodeValue`. That is: they are both null, or they have the same length and are character for character identical.

- The `attributes` `NamedNodeMaps` are equal. That is: they are both `null`, or they have the same length and for each node that exists in one map there is a node that exists in the other map and is equal, although not necessarily at the same index.
- The `childNodes` `NodeLists` are equal. That is: they are both `null`, or they have the same length and contain equal nodes at the same index. Note that normalization can affect equality; to avoid this, nodes should be normalized before being compared.

Calling the `.isEqualNode()` method on a node in the DOM will ask if that node is equal to the node that you pass it as a parameter. Below I exhibt a case of an two equal nodes and two non-identical nodes.

live code: http://jsfiddle.net/domenlightenment/xw68Q

```
<!DOCTYPE html>
<html lang="en">
<body>

<input type="text">
<input type="text">

<textarea>foo</textarea>
<textarea>bar</textarea>

<script>

//logs true, because they are exactly idential
var input = document.querySelectorAll('input');
console.log(input[0].isEqualNode(input[1]));

//logs false, because the child text node is not the same
var textarea = document.querySelectorAll('textarea');
console.log(textarea[0].isEqualNode(textarea[1]));

</script>
</body>
</html>
```

Notes

If you don't care about two nodes being exactly equal but instead want to know if two node references refer to the same node you can simply check it using the `===` opertor (i.e. `document.body === document.body`). This will tell us if they are identical but no equal.

1.19 Getting a reference to the `Document` from an `element` using `ownerDocument`

The `ownerDocument` property when called on a node returns a reference to the `Document` the node is contained within. In the code below I get a reference to the `Document` of the `<body>` in the HTML document and the `Document` node for the `<body>` element contained inside of the iframe.

live code: N/A

```
<!DOCTYPE html>
<html lang="en">
<body>

<iframe src="http://someFileServedFromServerOnSameDomain.html"></iframe>

<script>

//get the window.document that the <body> is contained within
console.log(document.body.ownerElement);

//get the window.document the <body> inside of the iframe is contained within
console.log(window.frames[0].document.body.ownerElement);

</script>
</body>
</html>
```

If `ownerDocument` is called on the `Document` node the value returned is `null`.

Chapter 2 - Document Nodes

2.1 `HTMLDocument` node overview

The `HTMLDocument` constructor (which inherits from `Document`) when instantiated represents specifically a `DOCUMENT_NODE` (i.e. `window.document`) in the DOM. To verify this we can simply ask which constructor was used in the creation of the `document` node object.

live code: http://jsfiddle.net/domenlightenment/qRAzL

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

console.log(window.document.constructor); //logs function HTMLDocument() { [native co
de] }
console.log(window.document.nodeType); //logs 9, which is a numeric key mapping to DO
CUMENT_NODE

</script>
</body>
</html>
```

The code above concludes that the `HTMLDocument` constructor function constructs the `window.document` node object and that this node is a `DOCUMENT_NODE` object.

Notes

Both `Document` and `HTMLDocument` constructors are typically instantiated by the browser when you load an HTML document.

2.2 `HTMLDocument` properties and methods (including inherited)

To get accurate information pertaining to the available properties and methods on an `HTMLDocument` node its best to ignore the specification and to ask the browser what is available. Examine the arrays created in the code below detailing the properties and methods available from an `HTMLDocument` node (a.k.a. `window.document`) object.

live code: http://jsfiddle.net/domenlightenment/jprPe

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

//document own properties
console.log(Object.keys(document).sort());

//document own properties & inherited properties
var documentPropertiesIncludeInherited = [];
for(var p in document){
        documentPropertiesIncludeInherited.push(p);
}
console.log(documentPropertiesIncludeInherited.sort());
```

```
//document inherited properties only
var documentPropertiesOnlyInherited = [];
for(var p in document){
        if(
                !document.hasOwnProperty(p)){documentPropertiesOnlyInherited.push(p);
        }
}
console.log(documentPropertiesOnlyInherited.sort());

</script>
</body>
</html>
```

The available properties are many even if the inherited properties were not considered. Below I've hand pick a list of noteworthy properties and methods for the context of this chapter.

- *doctype*
- *documentElement*
- *implementation*
- *activeElement*
- *body*
- *head*
- *title*
- *lastModified*
- *referrer*
- *URL*
- *defaultview*

For a complete list checkout the MDN documentation which covers the general methods and properties available to *HTMLDocument* objects.

Notes

The *HTMLDocument* node object is used to access a great deal of the methods and properties available for working with the DOM (i.e. *document.querySelectorAll()*). You will be seeing many of the properties not discussed in this chapter discussed in the appropriate chapter's following this chapter.

2.3 Getting general HTML document information (title, url, referrer, lastModified)

The `document` object provides access to some general information about the HTML document/DOM being loaded. In the code below I use the `document.title`, `document.URL`, `document.referrer`, and `document.lastModified` properties to gain some general information about the `document`. Based on the property name the returned values should be obvious.

live code: http://jsfiddle.net/domenlightenment/pX8Le

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

var d = document;
console.log('title = ' +d.title);
console.log('url = ' +d.URL);
console.log('referrer = ' +d.referrer);
console.log('lastModified = ' +d.lastModified);

</script>
</body>
</html>
```

2.4 *document* child nodes

`Document` nodes can contain one `DocumentType` node object and one `Element` node object. This should not be a surprise since HTML documents typically contain only one doctype (e.g. `<!DOCTYPE html>`) and one element (e.g. `<html lang="en">`). Thus if you ask for the children (e.g. `document.childNodes`) of the `Document` object you will get an array containing at the very least the documents doctype/DTD and `<html lang="en">` element. The code below showcases that `window.document` is a type of node object (i.e `Document`) with child nodes.

live code: http://jsfiddle.net/domenlightenment/UasKc

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

//This is the doctype/DTD
console.log(document.childNodes[0].nodeType); //logs 10, which is a numeric key mappi
ng to DOCUMENT_TYPE_NODE

//This is the <html> element
console.log(document.childNodes[1].nodeType); //logs 1, which is a numeric key mappin
```

```
g to ELEMENT_TYPE_NODE

</script>
</body>
</html>
```

Notes

Don't confuse the *window.document* object created from *HTMLDocument* constructor with the *Document* object. Just remember *window.document* is the starting point for the DOM interface. That is why *document.childNodes* contains child nodes.

If a comment node (not discussed in this book) is made outside of the *<html lang="en">* element then it will become a child node of the *window.document*. However having comment nodes outside of the <html> element can cause some buggy results in IE and also is a violation of the DOM specification.

2.5 *document* provides shortcuts to *<!DOCTYPE>*, *<html lang="en">*, *<head>*, and *<body>*

Using the properties listed below we can get a shortcut reference to the following nodes:

- *document.doctype* refers to *<!DOCTYPE>*
- *document.documentElement* refers to *<html lang="en">*
- *document.head* refers to *<head>*
- *document.body* refers to *<body>*

This is demonstrated in the code below.

live code: http://jsfiddle.net/domenlightenment/XsSTM

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

console.log(document.doctype); // logs DocumentType {nodeType=10, ownerDocument=docum
ent, ...}

console.log(document.documentElement); // logs <html lang="en">

console.log(document.head); // logs <head>

console.log(document.body); // logs <body>
```

```
</script>

</body>
</html>
```

Notes

The doctype or DTD is a `nodeType` of 10 or `DOCUMENT_TYPE_NODE` and should not be confused with the `DOCUMENT_NODE` (aka `window.document` constructed from `HTMLDocument()`). The doctype is constructed from the `DocumentType()` constructor.

In Safari, Chrome, and Opera the `document.doctype` does not appear in the `document.childNodes` list.

2.6 Detecting DOM specifications/features using `document.implementation.hasFeature()`

Its possible using `document.implementation.hasFeature()` to ask (boolean) the current document what feature and level the browser has implemented/supports. For example we can ask if the browser has implemented the core DOM level 3 specification by passing the name of the feature and the version to the `hasFeature()` method. In the code below I ask if the browser has implemented the Core 2.0 & 3.0 specification.

live code: http://jsfiddle.net/domenlightenment/TYYZ6

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

console.log(document.implementation.hasFeature('Core','2.0'));
console.log(document.implementation.hasFeature('Core','3.0'));

</script>
</body>
</html>
```

The following table defines the features (spec calls these modules) and versions that you can pass the `hasFeature()` method.

Feature	Supported Versions
Core	1.0, 2.0, 3.0

XML	1.0, 2.0, 3.0
HTML	1.0, 2.0
Views	2.0
StyleSheets	2.0
CSS	2.0
CSS2	2.0
Events	2.0, 3.0
UIEvents	2.0, 3.0
MouseEvents	2.0, 3.0
MutationEvents	2.0, 3.0
HTMLEvents	2.0
Range	2.0
Traversal	2.0
LS (Loading and saving between files and DOM trees synchronously)	3.0
LS-Asnc (Loading and saving between files and DOM trees asynchronously)	3.0
Validation	3.0

Notes

Don't trust `hasFeature()` alone you should also use capability detection in addition to `hasFeature()`.

Using the `isSupported` method implementation information can be gathered for a specific/selected node only (i.e. `element.isSupported(feature,version)`).

You can determince online what a user agent supports by visiting http://www.w3.org/2003/02/06-dom-support.html. Here you will find a table indicating what the browser loading the url claims to implement.

2.7 Get a reference to the focus/active node in the `document`

Using the `document.activeElement` we can quickly get a reference to the node in the document that is focused/active. In the code below, on page load, I am setting the focus of the document to the `<textarea>` node and then gaining a reference to that node using the `activeElement` property.

live code: http://jsfiddle.net/domenlightenment/N9npb

```
<!DOCTYPE html>
<html lang="en">
<body>
<textarea></textarea>

<script>

//set focus to <textarea>
document.querySelector('textarea').focus();

//get reference to element that is focused/active in the document
console.log(document.activeElement); //logs <textarea>

</script>
</body>
</html>
```

Notes

The focused/active element returns elements that have the ability to be focused. If you visit a web page in a browser and start hitting the tab key you will see focus shifting from element to element in that page that can get focused. Don't confuse the selection of nodes (highlight sections of the HTML page with mouse) with elements that get focus for the purpose of inputting something with keystrokes, spacebar, or mouse.

2.8 `document.defaultview` is a shortcut to the head/global object

You should be aware that the `defaultView` property is a shortcut to the JavaScript head object or what some refer to as the global object. The head object in a web browser is the `window` object and `defaultView` will point to this object in a JavaScript browser enviroment. The code below demonstrates the value of `defaultView` in a browser.

live code: http://jsfiddle.net/domenlightenment/QqK6Q

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

console.log(document.defaultView) //reference, head JS object. Would be window object
 in a browser.

</script>
</body>
</html>
```

If you are dealing with a DOM that is headless or an JavaScript enviroment that is not running in a web browser (i.e. node.js) this property can get you access to the head object scope.

Chapter 3 - Element Nodes

3.1 `HTML*Element` object overview

Elements in an html document all have a unique nature and as such they all have a unique JavaScript constructor that instantiates the element as a node object in a DOM tree. For example an `<a>` element is created as a DOM node from the `HTMLAnchorElement()` constructor. Below we verify that an anchor element is created from `HTMLAnchorElement()`.

live code: http://jsfiddle.net/domenlightenment/TgcNu

```
<!DOCTYPE html>
<html lang="en">
<body>

<a></a>

<script>

// grab <a> element node from DOM and ask for the name of the constructor that constr
ucted it
console.log(document.querySelector('a').constructor);
//logs function HTMLAnchorElement() { [native code] }

</script>
</body>
</html>
```

The point I am trying to express in the previous code example is that each element in the DOM is constructed from a unique JavaScript constructor function. The list below (no a complete list) should give you a good sense of the constructors used to create HTML elements.

- `HTMLHtmlElement`

- *HTMLHeadElement*
- *HTMLLinkElement*
- *HTMLTitleElement*
- *HTMLMetaElement*
- *HTMLBaseElement*
- *HTMLIsIndexElement*
- *HTMLStyleElement*
- *HTMLBodyElement*
- *HTMLFormElement*
- *HTMLSelectElement*
- *HTMLOptGroupElement*
- *HTMLOptionElement*
- *HTMLInputElement*
- *HTMLTextAreaElement*
- *HTMLButtonElement*
- *HTMLLabelElement*
- *HTMLFieldSetElement*
- *HTMLLegendElement*
- *HTMLUListElement*
- *HTMLOListElement*
- *HTMLDListElement*
- *HTMLDirectoryElement*
- *HTMLMenuElement*
- *HTMLLIElement*
- *HTMLDivElement*
- *HTMLParagraphElement*
- *HTMLHeadingElement*
- *HTMLQuoteElement*
- *HTMLPreElement*
- *HTMLBRElement*

- `HTMLBaseFontElement`
- `HTMLFontElement`
- `HTMLHRElement`
- `HTMLModElement`
- `HTMLAnchorElement`
- `HTMLImageElement`
- `HTMLObjectElement`
- `HTMLParamElement`
- `HTMLAppletElement`
- `HTMLMapElement`
- `HTMLAreaElement`
- `HTMLScriptElement`
- `HTMLTableElement`
- `HTMLTableCaptionElement`
- `HTMLTableColElement`
- `HTMLTableSectionElement`
- `HTMLTableRowElement`
- `HTMLTableCellElement`
- `HTMLFrameSetElement`
- `HTMLFrameElement`
- `HTMLIFrameElement`

Keep in mind each `HTML*Element` above inherits properteis and methods from `HTMLElement`, `Element`, `Node`, and `Object`.

3.2 `HTML*Element` object properties and methods (including inherited)

To get accurate information pertaining to the available properties and methods on an `HTML*Element` node its best to ignore the specification and to ask the browser what is available. Examine the arrays created in the code below detailing the properties and methods available from HTML element nodes.

live code: http://jsfiddle.net/domenlightenment/vZUHw

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#">Hi</a>

<script>

var anchor = document.querySelector('a');

//element own properties
console.log(Object.keys(anchor).sort());

//element own properties & inherited properties
var documentPropertiesIncludeInherited = [];
for(var p in document){
        documentPropertiesIncludeInherited.push(p);
}
console.log(documentPropertiesIncludeInherited.sort());

//element inherited properties only
var documentPropertiesOnlyInherited = [];
for(var p in document){
        if(!document.hasOwnProperty(p)){
                documentPropertiesOnlyInherited.push(p);
        }
}
console.log(documentPropertiesOnlyInherited.sort());

</script>
</body>
</html>
```

The available properties are many even if the inherited properties were not considered. Below I've hand pick a list of note worthy properties and methods (inherited as well) for the context of this chapter.

- *`createElement()`*
- *`tagName`*
- *`children`*
- *`getAttribute()`*
- *`setAttribute()`*
- *`hasAttribute()`*
- *`removeAttribute()`*
- *`classList()`*
- *`dataset`*

- `attributes`

For a complete list check out the MDN documentation which covers the general properties and methods available to most HTML elements.

3.3 Creating Elements

`Element` nodes are instantiated for us when a browser interputs an HTML document and a corresponding DOM is built based on the contents of the document. After this fact, its also possible to programaticlly create `Element` nodes using `createElement()`. In the code below I create a `<textarea>` node and then inject that node into the live DOM tree.

live code: http://jsfiddle.net/domenlightenment/d3Yvv

```
<!DOCTYPE html>
<html lang="en">
<body>
<script>

var elementNode = document.createElement('textarea'); //HTMLTextAreaElement() constru
cts <textarea>
document.body.appendChild(elementNode);

console.log(document.querySelector('textarea')); //verify it's now in the DOM

</script>
</body>
</html>
```

The value passed to the `createElement()` method is a string that specifices the type of element (aka `tagName`) to be created.

Notes

This value passed to createElement is changed to a lower-case string before the element is created.

3.4 Get the tag name of an element

Using the `tagName` property we can access the name of an element. The `tagName` property returns the same value that using `nodeName` would return. Both return the value in uppercase regardless of the case in the source HTML document.

Below we get the name of an `<a>` element in the DOM.

live code: http://jsfiddle.net/domenlightenment/YJb3W

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href="#">Hi</a>

<script>

console.log(document.querySelector('a').tagName); //logs A

//the nodeName property returns the same value
console.log(document.querySelector('a').nodeName); //logs A

</script>
</body>
</html>
```

3.5 Getting a list/collection of element attributes and values

Using the `attributes` property (inherited by element nodes from `Node`) we can get a collection of the `Attr` nodes that an element currently has defined. The list returned is a `NameNodeMap`. Below I loop over the attributes collection exposing each `Attr` node object contained in the collection.

live code: http://jsfiddle.net/domenlightenment/9gVQf

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href='#' title="title" data-foo="dataFoo" class="yes" style="margin:0;" foo="boo">
</a>

<script>

var atts = document.querySelector('a').attributes;

for(var i=0; i< atts.length; i++){
        console.log(atts[i].nodeName +'='+ atts[i].nodeValue);
}

</script>
</body>
```

```
</html>
```

Notes

The array returned from accessing the attributes property should be consider live. Meaning that its contents can be changed at anytime.

The array that is returned inherits from the *NameNodeMap* which provides methods to operate on the array such as *getNamtedItem()*, *setNamedItem()*, and *removeNamedItem()*. Operating on *attributes* with these methods should be secondary to using *getAttribute()*, *setAttribute()*, *hasAttribute()*, *removeAttribute()*. Its this authors opinion that dealing with Attr nodes is messy. The only merit in using the *attributes* is found only in its funcitonaly for get a list of attributes.

The *attributes* property is an array like collection and has a read only *length* property.

Boolean attributres (e.g. *<option selected>foo</option>*) show up in the *attributes* list but of course have no value unless you provide one (e.g. *<option selected="selected">foo</option>*).

3.6 Getting, Setting, & Removing an element's attribute value

The most consistent way to get, set, or remove an elements attribute value is to use the *getAttribute(), setAttribute()*, and *removeAttribute()* method. In the code below I demonstrate each of these methods for managing element attributes.

live code: http://jsfiddle.net/domenlightenment/wp7rq

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href='#' title="title" data-foo="dataFoo" style="margin:0;" class="yes" foo="boo">
</a>

<script>

var atts = document.querySelector('a');

//remove attributes
atts.removeAttribute('href');
atts.removeAttribute('title');
atts.removeAttribute('style');
atts.removeAttribute('data-foo');
atts.removeAttribute('class');
atts.removeAttribute('foo');

//set (really re-set) attributes
atts.setAttribute('href','#');
atts.setAttribute('title','title');
atts.setAttribute('style','margin:0;');
atts.setAttribute('data-foo','dataFoo');
atts.setAttribute('class','yes');
atts.setAttribute('foo','boo');

//get attributes
console.log(atts.getAttribute('href'));
console.log(atts.getAttribute('title'));
console.log(atts.getAttribute('style'));
console.log(atts.getAttribute('data-foo'));
console.log(atts.getAttribute('class'));
console.log(atts.getAttribute('foo'));

</script>
</body>
</html>
```

Notes

Use `removeAttribute()` instead of setting the attribute value to `null` or `''` using `setAttribute()`

3.7 Verifying an element has a specific attribute

The best way to determine (i.e. boolean) if an element has an attribute is to use the `hasAttribute()` method. Below I verify if the `<a>` has a `href`, `title`, `style`, `data-foo`, `class`, and `foo` attribute.

live code: http://jsfiddle.net/domenlightenment/hbCCE

```
<!DOCTYPE html>
<html lang="en">
<body>

<a href='#' title="title" data-foo="dataFoo" style="margin:0;" class="yes" foo></a>

<script>

var atts = document.querySelector('a');

console.log(
        atts.hasAttribute('href'),
        atts.hasAttribute('title'),
        atts.hasAttribute('style'),
        atts.hasAttribute('data-foo'),
        atts.hasAttribute('class'),
        atts.hasAttribute('foo') //Notice this is true regardless if a value is defin
ed
)
```

```
</script>
</body>
</html>
```

This method will return `true` if the element contains the attribute even if the attribute has no value. For example using `hasAttribute()` we can get a boolean response for boolean attributes. In the code example below we check to see if a checkbox is checked.

live code: http://jsfiddle.net/domenlightenment/tb6Ja

```
<!DOCTYPE html>
<html lang="en">
<body>

<input type="checkbox" checked></input>

<script>

var atts = document.querySelector('input');

console.log(atts.hasAttribute('checked')); //logs true

</script>
</body>
</html>
```

3.8 Getting a list of class attribute values

Using the `classList` property available on element nodes we can access a list (i.e. `DOMTokenList`) of class attribute values that is much easier to work with than a space-delimited string value returned from the `className` property. In the code below I contrast the use of `classList` with `className`.

live code: http://jsfiddle.net/domenlightenment/DLJEA

```
<!DOCTYPE html>
<html lang="en">
<body>

<div class="big brown bear"></div>

<script>

var elm = document.querySelector('div');

console.log(elm.classList); //big brown bear {0="big", 1="brown", 2="bear", length=3,
 ...}
console.log(elm.className); //logs 'big brown bear'

</script>
</body>
</html>
```

Notes

Given the `classList` is an array like collection it has a read only `length` property.

`classList` is read-only but can be modifyied using the `add()`, `remove()`, `contains()`, and `toggle()` methods

IE9 does not support `classList`. Support will land in IE10. Several polyfills are avaliable.

3.9 Adding & removing sub-values to a class attribute

Using the `classList.add()` and `classList.remove()` methods its extremely simple to edit the value of a class attribute. In the code below I demonstrated adding and removing class values.

live code: http://jsfiddle.net/domenlightenment/YVaUU

```
<!DOCTYPE html>
<html lang="en">
<body>
<div class="dog"></div>

<script>

var elm = document.querySelector('div');

elm.classList.add('cat');
elm.classList.remove('dog');
console.log(elm.className); //'cat'

</script>
</body>
</html>
```

3.10 Toggling a class attribute value

Using the `classList.toggle()` method we can toggle a sub-value of the class attribute. This allows us to add a value if its missing or remove a value if its already added. In the code below I toggle the `'visible'` value and the `'grow'` value. Which essentially means I remove `'visible'` and add `'grow'` to the class attribute value.

live code: http://jsfiddle.net/domenlightenment/uFp6J

```
<!DOCTYPE html>
<html lang="en">
<body>
<div class="visible"></div>

<script>

var elm = document.querySelector('div');

elm.classList.toggle('visible');
elm.classList.toggle('grow');
console.log(elm.className); //'grow'

</script>
</body>
</html>
```

3.11 Determining if a class attribute value contains a specific value

Using the `classList.contains()` method its possible to determine (boolean) if a class attribute value contains a specific sub-value. In the code below we test weather the `<div>` class attribute contains a sub-value of `brown`.

live code: http://jsfiddle.net/domenlightenment/njyaP

```
<!DOCTYPE html>
<html lang="en">
<body>
<div class="big brown bear"></div>

<script>

var elm = document.querySelector('div');

console.log(elm.classList.contains('brown')); //logs true

</script>
</body>
</html>
```

3.12 Getting & Setting data-* attributes

The `dataset` property of a element node provides an object containing all of the attributes of an element that starts with data-*. Because its a simply a JavaScript object we can manipulate `dataset` and have the element in the DOM reflect those changes

live code: http://jsfiddle.net/domenlightenment/ystgj

```
<!DOCTYPE html>
<html lang="en">
<body>

<div data-foo-foo="foo" data-bar-bar="bar"></div>

<script>

var elm = document.querySelector('div');

//get
console.log(elm.dataset.fooFoo); //logs 'foo'
console.log(elm.dataset.barBar); //logs 'bar'

//set
elm.dataset.gooGoo = 'goo';
console.log(elm.dataset); //logs DOMStringMap {fooFoo="foo", barBar="bar", gooGoo="go
o"}

//what the element looks like in the DOM
console.log(elm); //logs <div data-foo-foo="foo" data-bar-bar="bar" data-goo-goo="goo
">

</script>
</body>
</html>
```

Notes

`dataset` contains camel case versions of data attributes. Meaning `data-foo-foo` will be listed as the property `fooFoo` in the dataset `DOMStringMap` object. The `-` is replaced by camel casing.

Removing a data-* attribute from the DOM is as simple using the `delete` operator on a property of the `datset` (e.g. `delete dataset.fooFoo`)

`dataset` is not supported in IE9. A polyfill is avaliable. However, you can always just use getAttribute('data-foo'), removeAttribute('data-foo'), setAttribute('data-foo'), hasAttribute('data-foo').

Chapter 4 - Element Node Selecting

4.1 Selecting a specific element node

The most common methods for getting a reference to a single element node are:

- *querySelector()*
- *getElementById()*

In the code below I leverage both of these methods to select an element node from the HTML document.

live code: http://jsfiddle.net/domenlightenment/b4Rch

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
<li>Hello</li>
<li>big</li>
<li>bad</li>
<li id="last">world</li>
</ul>

<script>

console.log(document.querySelector('li').textContent); //logs Hello
console.log(document.getElementById('last').textContent); //logs world

</script>
</body>
</html>
```

The *getElementById()* method is pretty simple compared to the more robust *querySelector()* method. The *querySelector()* method permits a parameter in the form of a CSS selector syntax. Basically you can pass this method a CSS 3 selector (e.g. *'#score>tbody>tr>td:nth-of-type(2)'*) which it will use to select a single element in the DOM.

Notes

`querySelector()` will return the first node element found in the document based on the selector. For example, in the code example above we pass a selector that would select all the li's in CSS, but only the first one is returned.

`querySelector()` is also defined on element nodes. This allows for the method to limit (allows for context querying) its results to a specific vein of the DOM tree

4.2 Selecting/creating a list (aka `NodeList`) of element nodes

The most common methods for selecting/creating a list of nodes in an HTML document are:

- `querySelectorAll()`
- `getElementsByTagName()`
- `getElementsByClassName()`

Below we use all three of these methods to create a list of the `<li>` elements in the document.

live code: http://jsfiddle.net/domenlightenment/nT7Lr

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
<li class="liClass">Hello</li>
<li class="liClass">big</li>
<li class="liClass">bad</li>
<li class="liClass">world</li>
</ul>

<script>

//all of the methods below create/select the same list of <li> elements from the DOM
console.log(document.querySelectorAll('li'));
console.log(document.getElementsByTagName('li'));
console.log(document.getElementsByClassName('liClass'));

</script>
</body>
</html>
```

If its not clear the methods used in the code example above do not select a specific element, but instead creates a list (aka `NodeLists`) of elements that you can select from.

Notes

`NodeLists` created from `getElementsByTagName()` and `getElementsByClassName()` are considered live are will always reflect the state of the document even if the document is updated after the list is created/selected.

The `querySelectorAll()` method does not return a live list of elements. Meaning that the list created from `querySelectorAll()` is a snap shot of the document at the time it was created and is not reflective of the document as it changes. The list is static not live.

`querySelectorAll()`, `getElementsByTagName()`, and `getElementsByClassName` are also defined on element nodes. This allows for the method to limit its results to specific vein(s) of the DOM tree (e.g. `document.getElementById('header').getElementsByClassName('a')`).

I did not mention the `getElementsByName()` method as it not commonly leverage over other solutions but you should be aware of its existence for selecting form, img, frame, embed, and object elements from a document that all have the same name attribute value.

Passing either `querySelectorAll()` or `getElementsByTagName()` the string `'*'`, which generally means all, will return a list of all elements in the document.

Keep in mind that `childNodes` will also return a `NodeList` just like `querySelectorAll()`, `getElementsByTagName()`, and `getElementsByClassName`

The `NodeLists` are array like (but does not inherit array methods) lists/collections and have a read only `length` property

4.3 Selecting all immediate child element nodes

Using the `children` property from an element node we can get a list (aka [`HTMLCollection`]) of all the immediate children nodes that are element nodes. In the code below I use `children` to create a selection/list of all of the `<li>`'s contained wiithin the `<ul>`.

live code: http://jsfiddle.net/domenlightenment/svfRC

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
<li><strong>Hi</strong></li>
<li>there</li>
</ul>

<script>

var ulElement = document.querySelector('ul').children;

//logs a list/array of all immediate child element nodes
console.log(ulElement); //logs [<li>, <li>]
```

```
</script>
</body>
</html>
```

Notice that using *children* only gives us the immediate element nodes excluding any nodes (e.g. text nodes) that are not elements. If the element has no children then *children* will return an empty array-like-list.

Notes

HTMLCollection's contain elements in document order, that is they are placed in the array in the order the elements appear in the DOM

HTMLCollection's are live, which means any change to the document will be reflected dynamically in the collection

4.4 Contextual element selecting

The methods, *querySelector()*, *querySelectorAll()*, *getElementsByTagName()*, and *getElementsByClassName* are also defined on element nodes. This allows for these methods to limit its results to specific vein(s) of the DOM tree. Or said another, you can select a specific context in which you would like the methods to search for element nodes.

live code: http://jsfiddle.net/domenlightenment/fL6tV

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>
<ul>
<li class="liClass">Hello</li>
<li class="liClass">big</li>
<li class="liClass">bad</li>
<li class="liClass">world</li>
</ul>
</div>

<ul>
<li class="liClass">Hello</li>
</ul>

<script>

//select a div as the context to run the selecting methods only on the contents of th
e div
```

```
var div = document.querySelector('div');

console.log(div.querySelector('ul'));
console.log(div.querySelectorAll('li'));
console.log(div.getElementsByTagName('li'));
console.log(div.getElementsByClassName('liClass'));

</script>
</body>
</html>
```

These methods not only operate on the live dom but programatic DOM structures that are created in code as well.

live code: http://jsfiddle.net/domenlightenment/CCnva

```
<!DOCTYPE html>
<html lang="en">
<body>

<script>

//create DOM structure
var divElm = document.createElement('div');
var ulElm = document.createElement('ul');
var liElm = document.createElement('li');
liElm.setAttribute('class','liClass');
ulElm.appendChild(liElm);
divElm.appendChild(ulElm);

//use selecting methods on DOM structure
console.log(divElm.querySelector('ul'));
console.log(divElm.querySelectorAll('li'));
console.log(divElm.getElementsByTagName('li'));
console.log(divElm.getElementsByClassName('liClass'));

</body>
</html>
```

4.5 Pre-configured selections/lists of element nodes

You should be aware that there are some legacy, pre-configured arrays-like-lists, containing element nodes from an HTML document. Below I list a couple of these (not the complete list) that might be handy to be aware of.

- *document.all* - all elements in HTML document
- *document.forms* - all *<form>* elements in HTML document

- `document.images` - all `<img>` elements in HTML document
- `document.links` - all `<a>` elements in HTML document
- `document.scripts` - all `<script>` elements in HTML document

Notes

These pre-configured arrays of element nodes inherit from the HTMLCollection interface/object

`HTMLCollection`'s are live just like `NodeList`'s.

Oddly `document.all` is constucted from a `HTMLAllCollection` not an `HTMLCollection`

4.6 Verify an element will be selected using `matchesSelector()`

Using the `matchesSelector()` method we can determine if an element will match a selector string. For example say we want to determine if an `<li>` is the first child element of a `<ul>`. In the code example below I select the first `<li>` inside of the `<ul>` and then ask if that element matches the selector, `li:first-child`. Because it in fact does, the `matchesSelector()` method returns `true`.

live code: http://jsfiddle.net/domenlightenment/9RayM

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul>
<li>Hello</li>
<li>world</li>
</ul>

<script>

//fails in modern browser must use browser prefix moz, webkit, o, and ms
console.log(document.querySelector('li').matchesSelector('li:first-child')); //logs f
alse

//prefix moz
//console.log(document.querySelector('li').mozMatchesSelector('li:first-child'));

//prefix webkit
//console.log(document.querySelector('li').webkitMatchesSelector('li:first-child'));

//prefix o
//console.log(document.querySelector('li').oMatchesSelector('li:first-child'));
```

```
//prefix ms
//console.log(document.querySelector('li').msMatchesSelector('li:first-child'));

</script>
</body>
</html>
```

Notes

matchesSelector has not seen much love from the browsers as its usage is behind browser prefixes `mozMatchesSelector()`, `webkitMatchesSelector()`, `oMatchesSelector()`, `msMatchesSelector()`

Chapter 5 - Element Node Geometry & Scrolling Geometry

5.1 Element node size, offsets, and scrolling overview

DOM nodes are parsed and painted into visual shapes when viewing html documents in a web browser. Nodes, mostly element nodes, have a corresponding visual representation made viewable/visual by browsers. To inspect and in some cases manipulate the visual representation and gemometry of nodes programatically a set of API's exists and are specified in the CSSOM View Module. A subset of methods and properties found in this specification provide an API to determine the geometry (i.e. size & position using offset) of element nodes as well as hooks for manipulating scrollable nodes and getting values of scrolled nodes. This chapter breaks down these methods and properties.

Notes

Most of the properties (excluding `scrollLeft` & `scrollTop`) from the CSSOM View Module specification are read only and calculated each time they are accessed. In other words, the values are live

5.2 Getting an elements `offsetTop` and `offsetLeft` values relative to the `offsetParent`

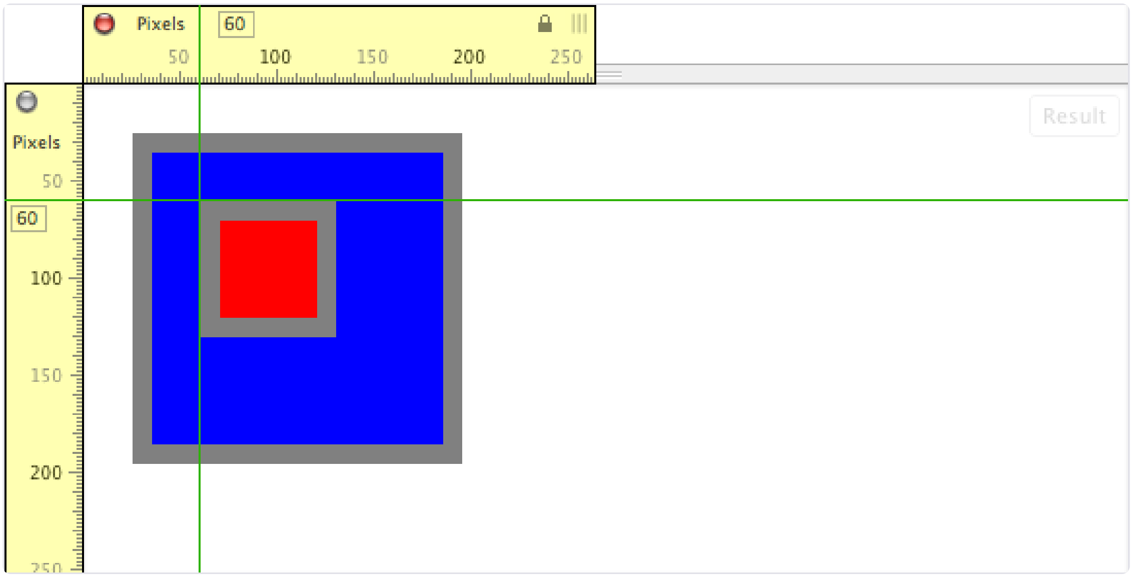
Using the properties `offsetTop` and `offsetLeft` we can get the offset pixel value of an element node from the `offsetParent`. These element node properties give us the distance in pixels from an elements outside top and left border to the inside top and left border of the `offsetParent`. The value of the `offsetParent` is determined by searching the nearest ancestor elements for an element that has a CSS position value not equal to static. If none are found then the `<body>` element or what some refer to as the "document" (as opposed to the browser viewport) is the `offsetParent` value. If during the ancestral search a `<td>`, `<th>`, or `<table>` element with a CSS position value of static is found then this becomes the value of `offsetParent`.

Lets verify that `offsetTop` and `offsetLeft` provide the values one might expect. The properties `offsetLeft` and `offsetTop` in the code below tell us that the `<div>` with an `id` of `red` is 60px's from the top and left of the `offsetParent` (i.e. the `<body>` element in this example).

live code: http://jsfiddle.net/domenlightenment/dj5h9

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
body{margin:0;}
#blue{height:100px;width:100px;background-color:blue;border:10px solid gray; padding:
25px;margin:25px;}
#red{height:50px;width:50px;background-color:red;border:10px solid gray;}
</style>
</head>
<body>

<div id="blue"><div id="red"></div></div>

<script>

var div = document.querySelector('#red');

console.log(div.offsetLeft); //logs 60
console.log(div.offsetTop); //logs 60
console.log(div.offsetParent); //logs <body>

</script>
</body>
</html>
```

Examine the following image of the web browser view to help aid your understanding of how the `offsetLeft` and `offsetTop` values are deteremined. The red `<div>` shown in the image is exactly 60 pixels from the `offsetParent`.



Notice I am measuring from the outside border of the red `<div>` element to the inside border of the `offsetParent` (i.e. `<body>`).

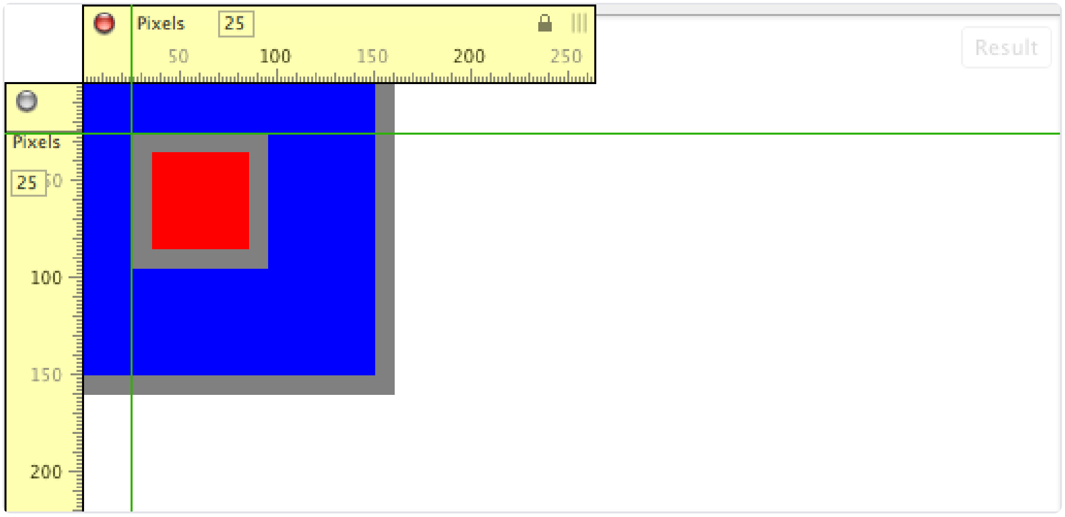
As previously mentioned If I was to change the blue `<div>` in the above code to have a position of absolute this would alter the value of the `offsetParent`. In the code below, absolutely positioning the blue `<div>` will cause the values returned from `offsetLeft` and `offsetTop` to report an offset (i.e. 25px's). This is because the offset parent is now the blue `<div>` and not the `<body>` .

live code: http://jsfiddle.net/domenlightenment/ft2ZQ

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
#blue{height:100px;width:100px;background-color:blue;border:10px solid gray; padding:
25px;margin:25px;position:absolute;}
#red{height:50px;width:50px;background-color:red;border:10px solid gray;}
</style>
</head>
<body>

<div id="blue"><div id="red"></div></div>

<script>

var div = document.querySelector('#red');

console.log(div.offsetLeft); //logs 25
console.log(div.offsetTop); //logs 25
console.log(div.offsetParent); //logs <div id="blue">

</script>
```

```
</body>
</html>
```

The image of the browser view shown below clarifies the new measurements returned from `offsetLeft` and `offsetTop` when the `offsetParent` is the blue `<div>`.



Notes

Many of the browsers break the outside border to inside border measurement when the `offsetParent` is the `<body>` and the `<body>` or `<html>` element has a visible margin, padding, or border value.

The `offsetParent`, `offsetTop`, and `offsetLeft` are extensions to the `HTMLelement` object.

5.3 Getting an elements top, right, bottom and left border edge offset relative to the viewport using `getBoundingClientRect()`

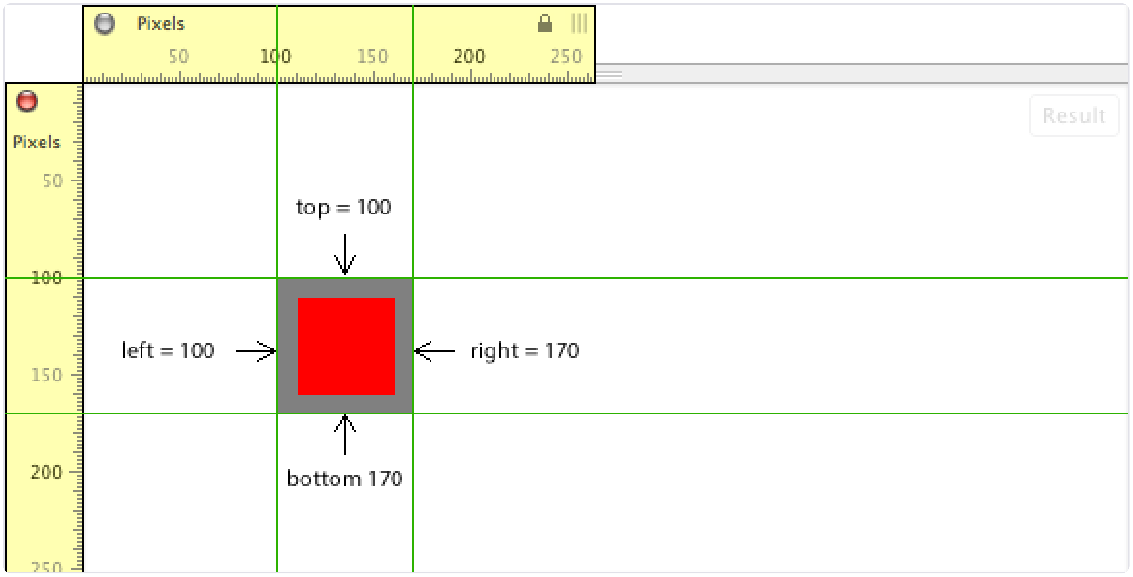
Using the `getBoundingClientRect()` method we can get the position of an elements outside border edges as its painted in the browser viewport relative to the top and left edge of the viewport. This means the left and right edge are measured from the outside border edge of an element to the left edge of the viewport. And the top and bottom edges are measured from the outside border edge of an element to the top edge of the viewport.

In the code below I create a 50px X 50px `<div>` with a 10px border and 100px margin. To get the distance in pixels from each border edge of the `<div>` I call the `getBoundingClientRect()` method on the `<div>` which returns an object containing a `top`, `right`, `bottom`, and `left` property.

live code: http://jsfiddle.net/domenlightenment/A3RN9

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
body{margin:0;}
div{height:50px;width:50px;background-color:red;border:10px solid gray;margin:100px;}
</style>
</head>
<body>

<div></div>

<script>

var divEdges = document.querySelector('div').getBoundingClientRect();

console.log(divEdges.top, divEdges.right, divEdges.bottom, divEdges.left); //logs '10
0 170 170 100'

</script>
</body>
</html>
```

The image below shows the browser rendered view of the above code with some added measurement indicators to show exactly how `getBoudingClientRect()` is calculated.



The `top` outside border edge of the `<div>` element is 100px from the top edge of the viewport. The `right` outside border edge of the element `<div>` is 170px from the left edge of the viewport. The `bottom` outside border edge of the element `<div>` is 170px from the top edge of the viewport. And the `left` outside border edge of the element `<div>` is 100px from the left edge of the viewport.

5.4 Getting an elements size (border + padding + content) in the viewport

The `getBoundingClientRect()` returns an object with a top, right, bottom, and left property/value but also with a height and width property/value. The `height` and `width` properties indicate the size of the element where the total size is derived by adding the content of the div, its padding, and borders together.

In the code below I get the size of the `<div>` element in the DOM using `getBoundingClientRect()`.

live code: http://jsfiddle.net/domenlightenment/PuXmL

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:25px;width:25px;background-color:red;border:25px solid gray;padding:25px;}
</style>
</head>
<body>

<div></div>

<script>

var div = document.querySelector('div').getBoundingClientRect();

console.log(div.height, div.width); //logs '125 125'
//because 25px border + 25px padding + 25 content + 25 padding + 25 border = 125

</script>
</body>
</html>
```

The exact same size values can also be found using from the `offsetHeight` and `offsetWidth` properties. In the code below I leverage these properties to get the same exact height and width values provided by `getBoundingClientRect()`.

live code: http://jsfiddle.net/domenlightenment/MSzL3

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:25px;width:25px;background-color:red;border:25px solid gray;padding:25px;}
</style>
</head>
<body>
```

```
<div></div>

<script>

var div = document.querySelector('div');

console.log(div.offsetHeight, div.offsetWidth); //logs '125 125'
//because 25px border + 25px padding + 25 content + 25 padding + 25 border = 125

</script>
</body>
</html>
```

5.5 Getting an elements size (padding + content) in the viewport excluding borders

The `clientWidth` and `clientHeight` properties return a total size of an element by adding together the content of the element and its padding excluding the border sizes. In the code below I use these two properties to get the height and width of an element including padding but excluding borders.

live code: http://jsfiddle.net/domenlightenment/bSrSb

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:25px;width:25px;background-color:red;border:25px solid gray;padding:25px;}
</style>
</head>
<body>

<div></div>

<script>

var div = document.querySelector('div');

console.log(div.clientHeight, div.clientWidth); //logs '75 75' because 25px padding +
 25 content + 25 padding = 75

</script>
</body>
</html>
```

5.6 Getting the size of the element being scrolled using `scrollHeight` and `scrollWidth`

The `scrollHeight` and `scrollWidth` properties simply give you the height and width of the node being scrolled. For example, open any HTML document that scrolls in a web browser and access these properties on the `<html>` (e.g. `document.documentElement.scrollWidth`) or `<body>` (e.g. `document.body.scrollWidth`) and you will get the total size of the HTML document being scrolled. Since we can apply, using CSS (i.e overflow:scroll), to elements lets look at a simpler code example. In the code below I make a `<div>` scroll a `<p>` element that is 1000px's x 1000px's. Accessing the `scrollHeight` and `scrollWidth` properties on the `<div>` will tell us that the element being scroll is 1000px's x 1000px's.

live code: http://jsfiddle.net/domenlightenment/9sZtZ

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
*{margin:0;padding:0;}
div{height:100px;width:100px; overflow:auto;}
p{height:1000px;width:1000px;background-color:red;}
</style>
</head>
<body>

<div><p></p></div>

<script>

var div = document.querySelector('div');

console.log(div.scrollHeight, div.scrollWidth); //logs '1000 1000'

</script>
</body>
</html>
```

Notes

If you need to know the height and width of the node inside a scrollable area when the node is smaller than the viewport of the scrollable area don't use `scrollHeight` and `scrollWidth` as this will give you the size of the viewport. If the node being scrolled is smaller than the scroll area then use `clientHeight` and `clientWidth` to determine the size of the node contained in the scrollable area.

5.7 Getting & Setting pixels scrolled from the top and left using `scrollTop` and `scrollLeft`

The `scrollTop` and `scrollLeft` properties are read-write properties that return the pixels to the left or top that are not currently viewable in the scrollable viewport due to scrolling. In the code below I setup a `<div>` that scrolls a `<p>` element.

live code: http://jsfiddle.net/domenlightenment/DqZYH

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:100px;width:100px;overflow:auto;}
p{height:1000px;width:1000px;background-color:red;}
</style>
</head>
<body>

<div><p></p></div>

<script>

var div = document.querySelector('div');

div.scrollTop = 750;
div.scrollLeft = 750;

console.log(div.scrollTop,div.scrollLeft); //logs '750 750'

</script>
</body>
</html>
```

I programatically scroll the `<div>` by setting the `scrollTop` and `scrollLeft` to 750. Then I get the current value of `scrollTop` and `scrollLeft`, which of course since we just set the value to 750 will return a value of 750. The 750 reports the number of pixels scroll and indicates 750 px's to the left and top are not viewable in the viewport. If it helps just think of these properties as the pixel measurements of the content that is not shown in the viewport to the left or top.

5.8 Scrolling an element into view using `scrollIntoView()`

By selecting a node contained inside a node that is scrollable we can tell the selected node to scroll into view using the `scrollIntoView()` method. In the code below I select the fifth `<p>` element contained in the scrolling `<div>` and call `scrollIntoView()` on it.

live code: http://jsfiddle.net/domenlightenment/SyeFZ

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{height:30px;width:30px; overflow:auto;}
p{background-color:red;}
</style>
</head>
<body>

<div>
<content>
<p>1</p>
<p>2</p>
<p>3</p>
<p>4</p>
<p>5</p>
<p>6</p>
<p>7</p>
<p>8</p>
<p>9</p>
<p>10</p>
</content>
</div>

<script>

//select <p>5</p> and scroll that element into view, I pass children '4' because its
a zero index array-like structure
document.querySelector('content').children[4].scrollIntoView(true);

</script>
</body>
</html>
```

By passing the `scrollIntoView()` method a parameter of `true` I am telling the method to scroll to the top of the element being scrolled too. The `true` parameter is however not needed as this is the default action performed by the method. If you want to scroll align to the bottom of the element pass a parameter of `false` to the `scrollIntoView()` method.

Chapter 6 - Element Node Inline Styles

6.1 Style Attribute (aka element inline CSS properties) Overview

Every HTML element has a style attribute that can be used to inline CSS properties specific to the element. In the code below I am accessing the `style` attribute of a `<div>` that contains several inline CSS properties.

live code: http://jsfiddle.net/domenlightenment/A4Aph

```
<!DOCTYPE html>
<html lang="en">
<body>

<div style="background-color:red;border:1px solid black;height:100px;width:100px;"></
div>

<script>

var divStyle = document.querySelector('div').style;

//logs CSSStyleDeclaration {0="background-color", ...}
console.log(divStyle);

 </script>
</body>
</html>
```

Notice that what is returned from the `style` property is a `CSSStyleDeclaration` object and not a string. Additionally note that only the elements inline styles (i.e. excluding the computed styles, computed styles being any styles that have cascaded from style sheets) are included in the `CSSStyleDeclartion` object.

6.2 Getting, setting, & removing individual inline CSS properties

Inline CSS styles are individually represented as a property (i.e. object property) on the `style` object.

This provides the interface for us to get, set, or remove individual CSS properties on an element. In the code below we set, get, and remove styles on a `<div>` by accesing CSS properties individually by property name.

live code: http://jsfiddle.net/domenlightenment/xNT85

```
<!DOCTYPE html>
<html lang="en">
<body>

<div></div>

<script>

var divStyle = document.querySelector('div').style;

//set
divStyle.backgroundColor = 'red';
divStyle.border = '1px solid black';
divStyle.width = '100px';
divStyle.height = '100px';

//get
console.log(divStyle.backgroundColor);
console.log(divStyle.border);
console.log(divStyle.width);
console.log(divStyle.height);

/*remove
divStyle.backgroundColor = '';
divStyle.border = '';
divStyle.width = '';
divStyle.height = '';
*/

</script>
</body>
</html>
```

Notes

The property names contained in the style object do not contain the normal hyphen that is used in CSS property names. The translation is pretty simple. Remove the hyphen and use camel case. (e.g. font-size = `fontSize` or background-image = `backgroundImage`). In the case where a css property name is a JavaScript keyword the javascript css property name is prefixed with "css" (e.g. float = `cssFloat`).

Short hand properties are available as properties as well. So you can not only set `margin`, but also `marginTop`.

Remember to include for any css property value that requires a unit of measure the appropriate unit (e.g. `style.width = '300px'; not style.width = '300';`). When a document is rendered in standards mode the unit of measure is require or it will be ignored. In quirksmode assumptions are made if not unit of measure is included.

The `CSSStyleDeclaration` object provides not only access to inidividual CSS properties, but also the `setPropertyValue(propertyName)`, `getPropertyValue(propertyName,value)`, and `removeProperty()` methods used to manipulate individual CSS properties on a element node. In the code below we set, get, and remove individual CSS properties on a `<div>` using these methods.

live code: http://jsfiddle.net/domenlightenment/X2DyX

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
</style>
</head>

<body>

<div style="background-color:green;border:1px solid purple;"></div>

<script>

var divStyle = document.querySelector('div').style;

//set
divStyle.setProperty('background-color','red');
divStyle.setProperty('border','1px solid black');
divStyle.setProperty('width','100px');
divStyle.setProperty('height','100px');

//get
console.log(divStyle.getPropertyValue('background-color'));
console.log(divStyle.getPropertyValue('border','1px solid black'));
console.log(divStyle.getPropertyValue('width','100px'));
console.log(divStyle.getPropertyValue('height','100px'));

/*remove
divStyle.removeProperty('background-color');
divStyle.removeProperty('border');
divStyle.removeProperty('width');
divStyle.removeProperty('height');
*/

</script>
</body>
</html>
```

Notes

Take notice that the property name is passed to the `setProperty()` and `getPropertyValue()` method using the css property name including a hyphen (e.g. `background-color` not `backgroundColor`).

For more detailed information about the `setProperty()`, `getPropertyValue()`, and `removeProperty()` as well as additional properties and methods have a look the documentation provided by Mozilla.

6.3 Getting, setting, & removing all inline CSS properties

Its possible using the `cssText` property of the `CSSStyleDeclaration` object as well as the `getAttribute()` and `setAttribute()` method to get, set, and remove the entire (i.e. all inline CSS properties) value of the style attribute using a JavaScript string. In the code below we get, set, and remove all inline CSS (as opposed to individually changing CSS proeprties) on a `<div>`.

live code: http://jsfiddle.net/domenlightenment/wSv8M

```
<!DOCTYPE html>
<html lang="en">
<body>

<div></div>

<script>

var div = document.querySelector('div');
var divStyle = div.style;

//set using cssText
divStyle.cssText = 'background-color:red;border:1px solid black;height:100px;width:10
0px;';
//get using cssText
console.log(divStyle.cssText);
//remove
divStyle.cssText = '';

//exactly that same outcome using setAttribute() and getAttribute()

//set using setAttribute
div.setAttribute('style','background-color:red;border:1px solid black;height:100px;wi
dth:100px;');
//get using getAttribute
console.log(div.getAttribute('style'));
//remove
div.removeAttribute('style');

</script>
</body>
</html>
```

Notes

If its not obvious you should note that replacing the `style` attribute value with a new string is the fastest way to make multiple changes to an elements style.

6.4 Getting an elements computed styles (i.e. actual styles including any from the cascade) using `getComputedStyle()`

The `style` property only contains the css that is defined via the style attribute. To get an elements css from the cascade (i.e. cascading from inline style sheets, external style sheets, browser style sheets) as well as its inline styles you can use `getComputedStyle()`. This method provides a read-only `CSSStyleDeclaration` object similar to `style`. In the code example below I demonstrate the reading of cascading styles, not just element inline styles.

live code: http://jsfiddle.net/domenlightenment/k3G5Q

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
div{
    background-color:red;
    border:1px solid black;
    height:100px;
    width:100px;
}
</style>
</head>

<body>

<div style="background-color:green;border:1px solid purple;"></div>

<script>

var div = document.querySelector('div');

//logs rgb(0, 128, 0) or green, this is an inline element style
console.log(window.getComputedStyle(div).backgroundColor);

//logs 1px solid rgb(128, 0, 128) or 1px solid purple, this is an inline element styl
e
console.log(window.getComputedStyle(div).border);

//logs 100px, note this is not an inline element style
console.log(window.getComputedStyle(div).height);

//logs 100px, note this is not an inline element style
console.log(window.getComputedStyle(div).width);

</script>
</body>
</html>
```

Make sure you note that `getComputedStyle()` method honors the CSS specificity hierarchy. For example in the code just shown the `backgroundColor` of the `<div>` is reported as green not red because inline styles are at the top of the specificity hierarchy thus its the inline `backgroundColor` value that is applied to the element by the browser and consider its final computed style.

Notes

No values can by set on a `CSSStyleDeclaration` object returned from `getComputedStyles()` its read only.

The `getComputedStyles()` method returns color values in the `rgb(#,#,#)` format regardless of how they were originally authored.

Shorthand properties are not computed for the `CSSStyleDeclaration` object you will have to use non-shorthand property names for property access (e.g. marginTop not margin).

6.5 Apply & remove css properties on an element using `class` & `id` attributes

Style rules defined in a inline style sheet or external style sheet can be added or removed from an element using the `class` and `id` attribute. This is a the most common pattern for manipulating element styles. In the code below, leveraging `setAttribute()` and `classList.add()`, inline style rules are applied to a `<div>` by setting the `class` and `id` attribute value. Using `removeAttribute()` and `classList.remove()` these CSS rules can be removed as well.

live code: http://jsfiddle.net/domenlightenment/BF9gM

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
.foo{
  background-color:red;
  padding:10px;
}
#bar{
  border:10px solid #000;
  margin:10px;
}
</style>
</head>
<body>

<div></div>

<script>

var div = document.querySelector('div');
```

```
//set
div.setAttribute('id','bar');
div.classList.add('foo');

/*remove
div.removeAttribute('id');
div.classList.remove('foo');
*/

</script>
</body>
</html>
```

Chapter 7 - Text Nodes

7.1 `Text` object overview

Text in an HTML document is represented by the results of the `Text()` constructor function, which produces text nodes. When an HTML document is parsed the text mixed in among the elements of an HTML page are converted to text nodes using `Text()`.

live code: http://jsfiddle.net/domenlightenment/kuz5Z

```
<!DOCTYPE html>
<html lang="en">
<body>

<p>hi</p>

<script>

//select 'hi' text node
var textHi = document.querySelector('p').firstChild

console.log(textHi.constructor); //logs Text()

//logs Text {textContent="hi", length=2, wholeText="hi", ...}
console.log(textHi);

</script>
</body>
</html>
```

The code above concludes that the `Text()` constructor function constructs the text node but keep in mind that `Text` inherits from `CharacterData`, `Node`, and `Object`.

7.2 `Text` object & properties

To get accurate information pertaining to the available properties and methods on an `Text` node its best to ignore the specification and to ask the browser what is available. Examine the arrays created in the code below detailing the properties and methods available from a text node.

live code: http://jsfiddle.net/domenlightenment/Wj3uS

```
<!DOCTYPE html>
<html lang="en">
<body>

<p>hi</p>

<script>
var text = document.querySelector('p').firstChild;

//text own properties
console.log(Object.keys(text).sort());

//text own properties & inherited properties
var textPropertiesIncludeInherited = [];
for(var p in text){
        textPropertiesIncludeInherited.push(p);
}
console.log(textPropertiesIncludeInherited.sort());

//text inherited properties only
var textPropertiesOnlyInherited = [];
for(var p in text){
        if(!text.hasOwnProperty(p)){
                textPropertiesOnlyInherited.push(p);
        }
}
console.log(textPropertiesOnlyInherited.sort());

</script>
</body>
</html>
```

The available properties are many even if the inherited properties were not considered. Below I've hand pick a list of note worthy properties and methods for the context of this chapter.

- `createTextNode()`

- *`textContent`*
- *`splitText()`*
- *`appendData()`*
- *`deleteData()`*
- *`insertData()`*
- *`replaceData()`*
- *`subStringData()`*
- *`normalize()`*
- *`data`*

7.3 White space creates *`Text`* nodes

When a DOM is contstructed either by the browser or by programmatic means text nodes are created from white space as well as from text characters. In the code below the second paragraph, conaining an empty space, has a child *`Text`* node while the first paragraph does not.

live code: http://jsfiddle.net/domenlightenment/YbtnZ

```
<!DOCTYPE html>
<html lang="en">
<body>

<p id="p1"></p>
<p id="p2"> </p>

<script>

console.log(document.querySelector('#p1').firstChild) //logs null
console.log(document.querySelector('#p2').firstChild.nodeName) //logs #text

</script>
</body>
</html>
```

Don't forget that white space and text characters in the DOM are typically represented by a text node. This of course means that carriage returns are considered text nodes. In the code below we log a carriage return highlighting the fact that this type of character is in fact a text node.

live code: http://jsfiddle.net/domenlightenment/9FEzq

```
<!DOCTYPE html>
<html lang="en">
<body>

<p id="p1"></p> //yes there is a carriage return text node before this comment, even
this comment is a node
<p id="p2"></p>

<script>

console.log(document.querySelector('#p1').nextSibling) //logs Text

</script>
</body>
</html>
```

The reality is if you can input the character or whitespace into an html document using a keyboard then it can potentially be interputed as a text node. If you think about, unless you minimze/compress the html document the average html page contains 100's of whitespace text nodes.

7.4 Creating & Injecting `Text` Nodes

`Text` nodes are created automatically for us when a browser interputs an HTML document and a corresponding DOM is built based on the contents of the document. After this fact, its also possible to programaticlly create `Text` nodes using `createTextNode()`. In the code below I create a text node and then inject that node into the live DOM tree.

live code: http://jsfiddle.net/domenlightenment/xC9q3

```
<!DOCTYPE html>
<html lang="en">
<body>

<div></div>

<script>

var textNode = document.createTextNode('Hi');
document.querySelector('div').appendChild(textNode);

console.log(document.querySelector('div').innerText); // logs Hi

</script>
</body>
</html>
```

Keep in mind that we can also inject text nodes into programmatically created DOM structures as well. In the code below I place the a text node inside of an `<p>` element before I inject it into the live DOM.

live code: http://jsfiddle.net/domenlightenment/PdatJ

```
<!DOCTYPE html>
<html lang="en">

<div></div>

<body>

<script>

var elementNode = document.createElement('p');
var textNode = document.createTextNode('Hi');
elementNode.appendChild(textNode);
document.querySelector('div').appendChild(elementNode);

console.log(document.querySelector('div').innerHTML); //logs <div>Hi</div>

</script>
</body>
</html>
```

7.5 Getting a `Text` node value with `.data` or `nodeValue`

The text value/data represented by a `Text` node can be extracted from the node by using the `.data` or `nodeValue` property. Both of these return the text contained in a `Text` node. Below I demostrate both of these to retrive the value contained in the `<div>`.

live code: http://jsfiddle.net/domenlightenment/dPLkx

```
<!DOCTYPE html>
<html lang="en">

<p>Hi, <strong>cody</strong></p><body>

<script>

console.log(document.querySelector('p').firstChild.data); //logs 'Hi,'
console.log(document.querySelector('p').firstChild.nodeValue); //logs 'Hi,'

</script>
</body>
</html>
```

Notice that the `<p>` contains two `Text` node and `Element` (i.e. `<strong>`)node. And that we are only getting the value of the first child node contained in the `<p>`.

Notes

Getting the length of a the chracters contained in a text node is as simple accessing the length proerty of the node itself or the actual text value/data of the node (i.e. `document.querySelector('p').firstChild.length`, `document.querySelector('p').firstChild.data.length`, `document.querySelector('p').firstChild.nodeValue.length`)

7.6 Maniputlating `Text` nodes with `appendData()`, `deleteData()`, `insertData()`, `replaceData()`, `subStringData()`

The `CharacterData` object that `Text` nodes inherits methods from provides the following methods for manipulating and extracting sub values from `Text` node values.

- `appendData()`
- `deleteData()`
- `insertData()`
- `replaceData()`
- `subStringData()`

Each of these are leverage in the code example below.

live code: http://jsfiddle.net/domenlightenment/B6AC6

```
<!DOCTYPE html>
<html lang="en">

<p>Go big Blue Blue<body>

<script>

var pElementText = document.querySelector('p').firstChild;

//add !
pElementText.appendData('!');
console.log(pElementText.data);

//remove first 'Blue'
pElementText.deleteData(7,5);
console.log(pElementText.data);
```

```
//insert it back 'Blue'
pElementText.insertData(7,'Blue ');
console.log(pElementText.data);

//replace first 'Blue' with 'Bunny'
pElementText.replaceData(7,5,'Bunny ');
console.log(pElementText.data);

//extract substring 'Blue Bunny'
console.log(pElementText.substringData(7,10));

</script>
</body>
</html>
```

Notes

These same manipulation and sub extraction methods can be leverage by `Comment` nodes

7.7 When mulitple sibling `Text` nodes occur

Typically, immediate sibling `Text` nodes do not occur because DOM trees created by browsers intelligently combines text nodes, however two cases exist that make sibling text nodes possible. The first case is rather obvious. If a text node contains an `Element` node (e.g. `<p>Hi, <strong>cody</strong> welcome!</p>`) than the text will be split into the proper node groupings. Its best to look at a code example as this might sound more complicted than it really is. In the code below the contents of the `<p>` element is not a single `Text` node it is in fact 3 nodes, a `Text` node, `Element` node, and another `Text` node.

live code: http://jsfiddle.net/domenlightenment/2ZCn3

```
<!DOCTYPE html>
<html lang="en">
<body>

<p>Hi, <strong>cody</strong> welcome!</p>

<script>

var pElement = document.querySelector('p');

console.log(pElement.childNodes.length); //logs 3

console.log(pElement.firstChild.data); // is text node or 'Hi, '
console.log(pElement.firstChild.nextSibling); // is Element node or <strong>
console.log(pElement.lastChild.data); // is text node or ' welcome!'
```

```
</script>
</body>
</html>
```

The next case occurs when we are programatically add `Text` nodes to an element we created in our code. In the code below I create a `<p>` element and then append two `Text` nodes to this element. Which results in sibling `Text` nodes.

live code: http://jsfiddle.net/domenlightenment/jk3Jn

```
<!DOCTYPE html>
<html lang="en">
<body>

<script>

var pElementNode = document.createElement('p');
var textNodeHi = document.createTextNode('Hi ');
var textNodeCody = document.createTextNode('Cody');

pElementNode.appendChild(textNodeHi);
pElementNode.appendChild(textNodeCody);

document.querySelector('div').appendChild(pElementNode);

console.log(document.querySelector('div p').childNodes.length); //logs 2

</script>
</body>
</html>
```

7.8 Remove markup and return all child `Text` nodes using `textContent`

The `textContent` property can be used to get all child text nodes, as well as to set the contents of a node to a specific `Text` node. When its used on a node to get the textual content of the node it will returned a concatenataed string of all text nodes contained with the node you call the method on. This functionality would make it very easy to extract all text nodes from an HTML document. Below I extract all of the text contained withing the `<body>` element. Notice that `textContent` gathers not just immediate child text nodes but all child text nodes no matter the depth of encapsulation inside of the node the method is called.

live code: N/A

```
<!DOCTYPE html>
<html lang="en">
<body>
<h1> Dude</h2>
<p>you <strong>rock!</strong></p>
<script>

console.log(document.body.textContent); //logs 'Dude you rock!' with some added white
 space

</script>
</body>
</html>
```

When `textContent` is used to set the text contained within a node it will remove all child nodes first, replacing them with a single `Text` node. In the code below I replace all the nodes inside of the `<div>` element with a single `Text` node.

live code: http://jsfiddle.net/domenlightenment/m766T

```
<!DOCTYPE html>
<html lang="en">
<body>
<div>
<h1> Dude</h2>
<p>you <strong>rock!</strong></p>
</div>
<script>

document.body.textContent = 'You don\'t rock!'
console.log(document.querySelector('div').textContent); //logs 'You don't rock!'

</script>
</body>
</html>
```

Notes

`textContent` returns `null` if used on the a document or doctype node.

`textContent` returns the contents from `<script>` and `<style>` elements

7.9 The difference between `textContent` & `innerText`

Most of the modern bowser, except Firefox, support a seeminly similiar property to `textContent` named `innerText`. However these properties are not the same. You should be aware of the following differences between `textContent` & `innerText`.

- `innerText` is aware of CSS. So if you have hidden text `innerText` ignores this text, whereas `textContent` will not
- Because `innerText` cares about CSS it will trigger a reflow, whereas `textContent` will not
- `innerText` ignores the `Text` nodes contained in `<script>` and `<style>` elements
- `innerText`, unlike `textContent` will normalize the text that is returned. Just think of `textContent` as returning exactly what is in the document with the markup removed. This will include white space, line breaks, and carriage returns
- `innerText` is considered to be non-standard and browser specific while `textContent` is implemented from the DOM specifications

If you you intend to use `innerText` you'll have to create a work around for Firefox.

7.10 Combine sibling `Text` nodes into one node using `normalize()`

Sibling `Text` nodes are typically only encountered when text is programaticly added to the DOM. To eliminate sibling `Text` nodes that contain no `Element` nodes we can use `normalize()`. This will concatenate sibling text nodes in the DOM into a single `Text` node. In the code below I create sibling text, append it to the DOM, then normalize it.

live code: http://jsfiddle.net/domenlightenment/LG9WR

```
<!DOCTYPE html>
<html lang="en">
<body>
<div></div>
<script>

var pElementNode = document.createElement('p');
var textNodeHi = document.createTextNode('Hi');
var textNodeCody = document.createTextNode('Cody');

pElementNode.appendChild(textNodeHi);
pElementNode.appendChild(textNodeCody);

document.querySelector('div').appendChild(pElementNode);

console.log(document.querySelector('p').childNodes.length); //logs 2

document.querySelector('div').normalize(); //combine our sibling text nodes

console.log(document.querySelector('p').childNodes.length); //logs 1
```

```
</script>
</body>
</html>
```

7.11 Splitting a text node using `splitText()`

When `splitText()` is called on a `Text` node it will alter the text node its being called on (leaving the text up to the offset) and return a new `Text` node that contains the text split off from the orginal text based on the offset. In the code below the text `Hey Yo!` is split after `Hey`.

live code: http://jsfiddle.net/domenlightenment/Tz5ce

```
<!DOCTYPE html>
<html lang="en">
<body>

<p>Hey Yo!</p>

<script>

//returns a new text node, taken from the DOM
console.log(document.querySelector('p').firstChild.splitText(4).data); //logs Yo!

//What remains in the DOM...
console.log(document.querySelector('p').firstChild.textContent); //logs Hey

</script>
</body>
</html>
```

Chapter 8 - DocumentFragment Nodes

8.1 `DocumentFragment` object overview

The creation and use of a `DocumentFragment` node provides a light weight document DOM that is external to the live DOM tree. Think of a `DocumentFragment` as an empty document template that acts just like the live DOM tree, but only lives in memory, and its child nodes can easily be manipulated in memory and then appended to the live DOM.

8.2 Creating `DocumentFragment`'s using `createDocumentFragment()`

In the code below a `DocumentFragment` is created using `createDocumentFragment()` and `<li>`'s are appended to it.

live code: http://jsfiddle.net/domenlightenment/6e3uX

```
<!DOCTYPE html>
<html lang="en">
<body>

<script>

var docFrag = document.createDocumentFragment();

["blue", "green", "red", "blue", "pink"].forEach(function(e) {
    var li = document.createElement("li");
    li.textContent = e;
    docFrag.appendChild(li);
});

console.log(docFrag.textContent); //logs bluegreenredbluepink

</script>
</body>
</html>
```

Using a `documentFragment` to create node structures in memory is extrememly efficent when it comes time to inject the `documentFragment` into live node structures.

You might wonder what is the advantage to using a `documentFragment` over simply creating (via `createElement()`) a `<div>` in memory and working within this `<div>` to create a DOM structure. The follow are the differences.

- A document fragment may contain any kind of node (except `<body>` or `<html>`) where as an element may not

- The document fragment itself, is not added to the DOM when you append a fragment. The contents of the node are.
- When a document fragment is appended to the DOM it transfers from the document fragment to the place its appended. Its no longer in memory in the place you created it.

8.3 Adding a `DocumentFragment` to the live DOM

By passing the `appendChild()` and `insertBefore()` node methods a `documentFragement` argument the child nodes of the `documentFragment` are transported as children nodes to the DOM node the methods are called on. Below we create a `documentfragment`, add some `<li>`'s to it, then append these new element nodes to the live DOM tree using `appendChild()`.

live code: http://jsfiddle.net/domenlightenment/Z2LpU

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul></ul>

<script>

var ulElm = document.queryselector('ul');
var docFrag = document.createDocumentFragment();

["blue", "green", "red", "blue", "pink"].forEach(function(e) {
    var li = document.createElement("li");
    li.textContent = e;
    docFrag.appendChild(li);
});

ulElm.appendChild(docFrag);

//logs <ul><li>blue</li><li>green</li><li>red</li><li>blue</li><li>pink</li></ul>
console.log(document.body.innerHTML);

</script>
</body>
</html>
```

Notes

Document fragments passed as arguments to inserting node methods will insert the entire child node structure ignoring the documentFragment node itself.

8.4 Using `innerHTML` on a `documentFragment` (i.e. HTML string to `Document`)

Creating a DOM structure in memory using node methods can be verbose and laboring. One way around this would be to created a `documentFragment`, append a `<div>` to this fragment because `innerHTML` does not work on document fragments, and then use the `innerHTML` property to update the fragment with a string of HTML. By doing this a DOM structure is crafted from the HTML string. In the code below I construct a DOM structure that I can then treat as a tree of nodes and not just a JavaScript string.

live code: http://jsfiddle.net/domenlightenment/4W9sH

```
<!DOCTYPE html>
<html lang="en">
<body>

<script>

//create a <div> and document fragment
var divElm = document.createElement('div');
var docFrag = document.createDocumentFragment();

//append div to document fragment
docFrag.appendChild(divElm);

//create a DOM structure from a string
docFrag.querySelector('div').innerHTML = '<ul><li>foo</li><li>bar</li></ul>';

//the string becomes a DOM structure I can call methods on like querySelectorAll()
//Just don't forget the DOM structure is wrapped in a <div>
console.log(docFrag.querySelectorAll('li').length); //logs 2

</script>
</body>
</html>
```

When it comes time to append a DOM structure created using a `documentFragment` and `<div>` you'll want to append the structure skipping the injection of the `<div>`.

live code: http://jsfiddle.net/domenlightenment/kkyKJ

```
<!DOCTYPE html>
<html lang="en">
<body>

<div></div>

<script>
```

```
//create a <div> and document fragment
var divElm = document.createElement('div');
var docFrag = document.createDocumentFragment();

//append div to document fragment
docFrag.appendChild(divElm);

//create a DOM structure from a string
docFrag.querySelector('div').innerHTML = '<ul><li>foo</li><li>bar</li></ul>';

//append, starting with the first child node contained inside of the <div>
document.querySelector('div').appendChild(docFrag.querySelector('div').firstChild);

//logs <ul><li>foo</li><li>bar</li></ul>
console.log(document.querySelector('div').innerHTML);

</script>
</body>
</html>
```

Notes

In addtion to `DocumentFragment` we also have `DOMParser` to look forward too. `DOMParser` can parse HTML source stored in a string into a DOM Document. It's only supported in Firefox as of today, but a polyfill is avaliable.

8.5 Leaving a fragments containing nodes in memory by cloning

When appending a `documentFragement` the nodes contained in the fragement are moved from the fragement to the structure you are appending too. To leave the contents of a fragement in memory, so the nodes remain after appending simply clone, using `cloneNode()`, the `documentFragment` when appending. In the code below instead of tranporting the `<li>`'s from the document fragment I clone the
`<li>`'s, which keeps the `<li>`'s being clonded in memory inside of the `documentFragment` node.

live code: http://jsfiddle.net/domenlightenment/bcJGS

```
<!DOCTYPE html>
<html lang="en">
<body>

<ul></ul>

<script>

//create ul element and document fragment
var ulElm = document.querySelector('ul');
var docFrag = document.createDocumentFragment();
```

```
//append li's to document fragment
["blue", "green", "red", "blue", "pink"].forEach(function(e) {
    var li = document.createElement("li");
    li.textContent = e;
    docFrag.appendChild(li);
});

//append cloned document fragment to ul in live DOM
ulElm.appendChild(docFrag.cloneNode(true));

//logs <li>blue</li><li>green</li><li>red</li><li>blue</li><li>pink</li>
console.log(document.querySelector('ul').innerHTML);

//logs [li,li,li,li,li]
console.log(docFrag.childNodes);

</script>
</body>
</html>
```

Chapter 9 - CSS Style Sheets & CSS rules

9.1 CSS Style sheet overview

A style sheet is added to an HTML document by either using the `HTMLLinkElement` node (i.e. `<link href="stylesheet.css" rel="stylesheet" type="text/css">`) to include an external style sheet or the `HTMLStyleElement` node (i.e. `<style></style>`) to define a style sheet inline. In the HTML document below both of these `Element` node's are in the DOM and I verify which constructor, constructs these nodes.

live code: http://jsfiddle.net/domenlightenment/yPYyC

```
<!DOCTYPE html>
<html lang="en">
<head>

<link id="linkElement" href="http://yui.yahooapis.com/3.3.0/build/cssreset/reset-min.
css" rel="stylesheet" type="text/css">
```

```
<style id="styleElement">
body{background-color:#fff;}
</style>

</head>
<body>

<script>

//logs function HTMLLinkElement() { [native code] }
console.log(document.querySelector('#linkElement').constructor);

//logs function HTMLStyleElement() { [native code] }
console.log(document.querySelector('#styleElement').constructor);

</script>
</body>
</html>
```

Once a style sheet is added to an HTML document its represented by the `CSSStylesheet` object. Each CSS rule (e.g. `body{background-color:red;}`) inside of a style sheet is represent by a `CSSStyleRule` object. In the code below I verify which constructor constructed the style sheet and each CSS rule (selector & its css properties and values) in the style sheet.

live code: http://jsfiddle.net/domenlightenment/UpLzm

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
body{background-color:#fff;}
</style>

</head>
<body>

<script>

//logs function CSSStyleSheet() { [native code] } because this object is the styleshe
et itself
console.log(document.querySelector('#styleElement').sheet.constructor);

//logs function CSSStyleRule() { [native code] } because this object is the rule insi
de of the style sheet
console.log(document.querySelector('#styleElement').sheet.cssRules[0].constructor);

</script>
</body>
</html>
```

Keep in mind that selecting the element that includes the style sheet (i.e. `<link>` or `<style`>) is not the same as accessing the actual object (`CSSStyleSheet`) that represents the style sheet itself.

9.2 Accessing all style sheets (i.e. `CSSStylesheet` objects) in the DOM

`document.styleSheets` gives access to a list of all style sheet objects (aka `CSSStylesheet`) explicitly linked (i.e. `<link>`) or embedded (i.e. `<style>`) in an HTML document. In the code below `styleSheets` is leverage to gain access to all of the style sheets contained in the document.

live code: N/A

```
<!DOCTYPE html>
<html lang="en">
<head>

<link href="http://yui.yahooapis.com/3.3.0/build/cssreset/reset-min.css" rel="stylesh
eet" type="text/css">

<style>
body{background-color:red;}
</style>

</head>
<body>

<script>

console.log(document.styleSheets.length); //logs 2
console.log(document.styleSheets[0]); // the <link>
console.log(document.styleSheets[1]); // the <style>

</script>
</body>
</html>
```

Notes

`styleSheet` is live just like other node lists

The `length` property returns the number of stylesheets contained in the list starting at 0 index (i.e. `document.styleSheets.length`)

The style sheets included in a `styleSheets` list typically includes any style sheets created using the `<style>` element or using a `<link>` element where `rel` is set to `"stylesheet"`

In addtion to using `styleSheets` to access a documents styles sheets its also possible to access a style sheet in an HTML document by first selecting the element in the DOM (`<style>` or `<link>`) and using the `.sheet` property to gain access to the `CSSStyleSheet` object. In the code below I access the style sheets in the HTML docment by first selecting the element used to include the style sheet and then leveraging the `sheet` property.

live code: http://jsfiddle.net/domenlightenment/jFwKw

```
<!DOCTYPE html>
<html lang="en">
<head>

<link id="linkElement" href="http://yui.yahooapis.com/3.3.0/build/cssreset/reset-min.
css" rel="stylesheet" type="text/css">

<style id="styleElement">
body{background-color:#fff;}
</style>

</head>
<body>

<script>

//get CSSStylesheeet object for <link>
console.log(document.querySelector('#linkElement').sheet); //same as document.styleSh
eets[0]

//get CSSSstylesheet object for <style>
console.log(document.querySelector('#styleElement').sheet); //same as document.styleS
heets[1]

</script>
</body>
</html>
```

9.3 `CSSStyleSheet` properties and methods

To get accurate information pertaining to the available properties and methods on an `CSSStyleSheet` node its best to ignore the specification and to ask the browser what is available. Examine the arrays created in the code below detailing the properties and methods available from a `CSSStyleSheet` node.

live code: http://jsfiddle.net/domenlightenment/kNyL2

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<style id="styleElement">
body{background-color:#fff;}
</style>

</head>
<body>

<script>

var styleSheet = document.querySelector('#styleElement').sheet;

//text own properties
console.log(Object.keys(styleSheet).sort());

//text own properties & inherited properties
var styleSheetPropertiesIncludeInherited = [];
for(var p in styleSheet){
        styleSheetPropertiesIncludeInherited.push(p);
}
console.log(styleSheetPropertiesIncludeInherited.sort());

//text inherited properties only
var styleSheetPropertiesOnlyInherited = [];
for(var p in styleSheet){
        if(!styleSheet.hasOwnProperty(p)){
                styleSheetPropertiesOnlyInherited.push(p);
        }
}
console.log(styleSheetPropertiesOnlyInherited.sort());

</script>
</body>
</html>
```

A `CSSStyleSheet` object accessed from a `styleSheets` list or via the `.sheet` property has the following properties and methods:

- `disabled`
- `href`
- `media`
- `ownerNode`
- `parentStylesheet`
- `title`
- `type`
- `cssRules`
- `ownerRule`

- *deleteRule*
- *inserRule*

Notes

href, *media*, *ownerNode*, *parentStylesheet*, *title*, and *type* are read only properties, you can't set its value using these properteis

9.4 *CSSStyleRule* overview

A *CSSStyleRule* object represents each CSS rule contained in a style sheet. Basicly a *CSSStyleRule* is the interface to the CSS properties and values attached to a selector. In the code below we programaticlly access the details of each rule contained in the inline style sheet by accessing the *CSSStyleRule* object that represents the CSS rule in the style sheet.

live code: http://jsfiddle.net/domenlightenment/fPVS8

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
body{background-color:#fff;margin:20px;} /*this is a css rule*/
p{line-height:1.4em; color:blue;} /*this is a css rule*/
</style>

</head>
<body>

<script>

var sSheet = document.querySelector('#styleElement');

console.log(sSheet.cssRules[0].cssText); //logs "body { background-color: red; margin
: 20px; }"
console.log(sSheet.cssRules[1].cssText); //logs "p { line-height: 1.4em; color: blue;
 }"

</script>
</body>
</html>
```

9.5 `CSSStyleRule` properties and methods

To get accurate information pertaining to the available properties and methods on an `CSSStyleRule` node its best to ignore the specification and to ask the browser what is available. Examine the arrays created in the code below detailing the properties and methods available from a `CSSStyleRule`node.

live code: http://jsfiddle.net/domenlightenment/hCX3U

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
body{background-color:#fff;}
</style>

</head>
<body>

<script>

var styleSheetRule = document.querySelector('#styleElement').sheet.cssRule;

//text own properties
console.log(Object.keys(styleSheetRule).sort());

//text own properties & inherited properties
var styleSheetPropertiesIncludeInherited = [];
for(var p in styleSheetRule){
        styleSheetRulePropertiesIncludeInherited.push(p);
}
console.log(styleSheetRulePropertiesIncludeInherited.sort());

//text inherited properties only
var styleSheetRulePropertiesOnlyInherited = [];
for(var p in styleSheetRule){
        if(!styleSheetRule.hasOwnProperty(p)){
                styleSheetRulePropertiesOnlyInherited.push(p);
        }
}
console.log(styleSheetRulePropertiesOnlyInherited.sort());

</script>
</body>
</html>
```

Scripting the rules (e.g. `body{background-color:red;}`) contained inside of a style sheet is made possible by the `CSSrule` object. This object provides the following properties:

- `cssText`

- *parentRule*
- *parentStylesSheet*
- *selectorText*
- *style*
- *type*

9.6 Getting a list of CSS Rules in a style sheet using *CSSRules*

As previously discussed the *styleSheets* list provides a list of style sheets contained in a document. The *CSSRules* list provides a list (aka *CSSRulesList*) of all the CSS rules (i.e. *CSSStyleRule* objects) in a specific style sheet. The code below logs a *CSSRules* list to the console.

live code: http://jsfiddle.net/domenlightenment/qKqhJ

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
body{background-color:#fff;margin:20px;}
p{line-height:1.4em; color:blue;}
</style>

</head>
<body>

<script>

var sSheet = document.querySelector('#styleElement').sheet;

//array like list containing all of the CSSrule objects repreesenting each CSS rule i
n the style sheet
console.log(sSheet.cssRules);

console.log(sSheet.cssRules.length); //logs 2

//rules are index in a CSSRules list starting at a 0 index
console.log(sSheet.cssRules[0]); //logs first rule
console.log(sSheet.cssRules[1]); //logs second rule

</script>
</body>
</html>
```

9.7 Inserting & deleting CSS rules in a style sheet using `.insertRule()` and `.deleteRule()`

The `insertRule()` and `deleteRule()` methods provided the ability to programatically manipulate the CSS rules in a style sheet. In the code below I use `insertRule()` to add the css rule `p{color:red}` to the inline style sheet at index 1. Remeber the css rules in a style sheet are numerical index starting at 0. So by inserting a new rule at index 1 the current rule at index 1 (i.e. `p{font-size:50px;}`) is push to index 2.

live code: http://jsfiddle.net/domenlightenment/T2jzJ

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
p{line-height:1.4em; color:blue;} /*index 0*/
p{font-size:50px;} /*index 1*/
</style>

</head>
<body>

<p>Hi</p>

<script>

//add a new CSS rule at index 1 in the inline style sheet
document.querySelector('#styleElement').sheet.insertRule('p{color:red}',1);

//verify it was added
console.log(document.querySelector('#styleElement').sheet.cssRules[1].cssText);

//Delete what we just added
document.querySelector('#styleElement').sheet.deleteRule(1);

//verify it was removed
console.log(document.querySelector('#styleElement').sheet.cssRules[1].cssText);

</script>
</body>
</html>
```

Deleting or removing a rule is as simple as calling `deleteRule()` method on a style sheet and passing it the index of the rule in the style sheet to be deleted.

Notes

Inserting and deleting rules is not a common practice given the difficulty around managing the cascaade and using a numeric indexing system to update a style sheet (i.e. determining at what index a style is located without previewing the contents of the style sheet itself.). Its much simpler working with CSS rules in CSS and HTML files before they are served to a client than programaticlly altering them in the client after the fact.

9.8 Editing the value of a `CSSStyleRule` using the `.style` property

Just like the `.style` property that facilitates the manipulation of inline styles on element nodes there is a also `.style` property for `CSSStyleRule` objects that orchestrates the same manipulation of styles in style sheets. In the code below I levereage the `.style` property to set and get the value of css rules contained in the inline style sheet.

live code: http://jsfiddle.net/domenlightenment/aZ9CQ

```
<!DOCTYPE html>
<html lang="en">
<head>

<style id="styleElement">
p{color:blue;}
strong{color:green;}
</style>

</head>
<body>

<p>Hey <strong>Dude!</strong></p>

<script>

var styleSheet = document.querySelector('#styleElement').sheet;

//Set css rules in stylesheet
styleSheet.cssRules[0].style.color = 'red';
styleSheet.cssRules[1].style.color = 'purple';

//Get css rules
console.log(styleSheet.cssRules[0].style.color); //logs 'red'
console.log(styleSheet.cssRules[1].style.color); //logs 'purple'

</script>
</body>
</html>
```

9.9 Creating a new inline CSS style sheets

To craft a new style sheet on the fly after an html page is loaded one only has to create a new `<style>` node, add CSS rules using `innerHTML` to this node, then append the `<style>` node to the HTML document. In the code below I programatily craft a style sheet and add the `body{color:red}` CSS rule to the style sheet, then append the stylesheet to the DOM.

live code: http://jsfiddle.net/domenlightenment/bKXAk

```
<!DOCTYPE html>
<html lang="en">
<head></head>
<body>

<p>Hey <strong>Dude!</strong></p>

<script>

var styleElm = document.createElement('style');
styleElm.innerHTML = 'body{color:red}';

//notice markup in the document changed to red from our new inline stylesheet
document.querySelector('head').appendChild(styleElm);

</script>
</body>
</html>
```

9.10 Programatically adding external style sheets to an HTML document

To add a CSS file to an HTML document programatically a `<link>` element node is created with the appropriate attributes and then the `<link>` element node is appended to the DOM. In the code below I programatically include an external style sheet by crafting a new `<link>` element and appending it to the DOM.

live code: http://jsfiddle.net/domenlightenment/dtwgC

```
<!DOCTYPE html>
<html lang="en">
<head></head>
<body>

<script>

//create & add attributes to <link>
var linkElm = document.createElement('link');
linkElm.setAttribute('rel', 'stylesheet');
linkElm.setAttribute('type', 'text/css');
linkElm.setAttribute('id', 'linkElement');
linkElm.setAttribute('href', 'http://yui.yahooapis.com/3.3.0/build/cssreset/reset-min
.css');

//Append to the DOM
document.head.appendChild(linkElm);

//confrim its addition to the DOM
console.log(document.querySelector('#linkElement'));

</script>
</body>
</html>
```

9.11 Disabling/Enabling style sheets using `disabled` property

Using the `.disabled` property of a `CSSStyleSheet` object its possible to enable or disabled a style sheet. In the code below we access the current disabled value of each style sheet in the document then proceed to disabled each style sheet leveraging the `.disabled` property.

live code: http://jsfiddle.net/domenlightenment/L952Z

```
<!DOCTYPE html>
<html lang="en">
<head>

<link id="linkElement" href="http://yui.yahooapis.com/3.3.0/build/cssreset/reset-min.
css" rel="stylesheet" type="text/css">

<style id="styleElement">
body{color:red;}
</style>

</head>
<body>

<script>

//Get current boolean disabled value
console.log(document.querySelector('#linkElement').disabled); //log 'false'
console.log(document.querySelector('#styleElement').disabled); //log 'false'

//Set disabled value, which of courese disabled all styles for this document
document.document.querySelector('#linkElement').disabled = true;
document.document.querySelector('#styleElement').disabled = true;

</script>
</body>
</html>
```

Notes

Disabled is not an avaliable attributre of a <link> or <style> element according to the specification. Trying to add this as an attribute in the HTML document itself will fail (and likley cause parsing errors where styles are ignored) in the majority of modern browsers in use today.

Chapter 10 - JavaScript in the DOM

10.1 Inserting & executing JavaScript overview

JavaScript can be inserted in to an HTML document in a modern way by including external JavaScript files or writing page level inline JavaScript, which is basically the contents of an external JavaScript file literally embed in the HTML page as a text node. Don't confuse element inline JavaScript contained in attribute event handlers (i.e. `<div onclick="alert('yo')"></div>`) with page inline JavaScript (i.e. `<script>alert('hi')</script>`).

Both methods of inserting JavaScript into an HTML document require the use of a `<script>` element node. The `<script>` element can contain JavaScript code or can be used to link to external JavaScript files using the `src` attribute. Both methods are explored in the code example below.

live code: http://jsfiddle.net/domenlightenment/g6T5F

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- external, cross domain JavaScript include -->
<script src="http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.3/underscore-min
.js"></script>

<!-- page inline JavaScript -->
<script>
console.log('hi');
</script>

</body>
```

```
</html>
```

Notes

Its possible to insert and execute JavaScript in the DOM by placing JavaScript in an element attribute event handler (i.e. `<div onclick="alert('yo')"></div>`) and using the `javascript:` protocol (e.g. `<a href="javascript:alert('yo')"></a>`) but this is no longer considered a modern practice.

Trying to include an external JavaScript file and writing page inline JavaScript using the same `<script>` element will result in the page inline JavaScript being ignored and the exterenal JavaScript file being downloaded and exectued

All JavaScript regardless of how its inserted in the DOM will be interpureted from the top to bottom

Self-closing scripts tags (i.e. `<script src="" />`) should be avoid unless you are rocking some old school XHTML

The `<script>` element does not have any required attributes but offers the follow optional attribures: `async`, `charset`, `defer`, `src`, and `type`

Page inline JavaScript produces a text node. Which permits the usage of `innerHTML` and `textContent` to retrieve the contents of a line `<script>`. However, appending a new text node made up of JavaScript code to the DOM after the browser has already parsed the DOM will not execute the new JavaScript code. It simply replaces the text.

If JavaScript code contains the string `'</script>'` you will have to escape the closing `'/'` with `'<\/script>'` so that the parser does not think this is the real closing `</script>` element

10.2 JavaScript is parsed synchronously by default

By default when the DOM is being parsed and it encounters a `<script>` element it will stop parsing the document, block any further rendering & downloading, and exectue the JavaScript. Because this behavior is blocking and does not permit parallel parsing of the DOM or exection of JavaScriopt its consider to be synchronous. If the JavaScript is external to the html document the blocking is exacerbated because the JavaScript must first be downloaed before it can be parsed. In the code example below I comment what is occuring during browser rendering when the browser encoutners several `<script>` elements in the DOM.

live code: http://jsfiddle.net/domenlightenment/rF3Lh

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- stop document parsing, block document parsing, load js, exectue js, then resume
document parsing... -->
```

```
<script src="http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.3/underscore-min
.js"></script><!-- stop document parsing, block document parsing, exectue js, then re
sume document parsing... --><script>console.log('hi');</script>


</body>
</html>
```

You should make note of the differences between an inline script's and external scripts as it pertains to the loading phase.

Notes

The default blocking nature of a `<script>` element can have a significant effect on the perfomrance & percived performance of the visual rendering of a HTML web page. If you have a couple of script elements at the start of an html page nothing else is happening (e.g. DOM parsing & resource loading) until each one is downloaed and executed sequentially.

10.3 Defering the downloading & exectuion of external JavaScript using `defer`

The `<script>` element has an attribute called `defer` that will defer the blocking, downloading, and executing of an external JavaScript file until the browser has parsed the closing `<html>` node. Using this attribute simply defers what normally occurs when a web browser encoutners a `<script>` node. In the code below I defer each external JavaScript file until the final `<html>` is encountered.

live code: http://jsfiddle.net/domenlightenment/HDegp

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- defer, don't block just ignore this until the <html> element node is parsed -->
<script defer src="http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.3/undersco
re-min.js"></script>

<!-- defer, don't block just ignore this until the <html> element node is parsed -->
<script defer src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.min.js">
</script>

<!-- defer, don't block just ignore this until the <html> element node is parsed -->
<script defer src="http://cdnjs.cloudflare.com/ajax/libs/jquery-mousewheel/3.0.6/jque
ry.mousewheel.min.js"></script>

<script>
//We know that jQuery is not avaliable because this occurs before the closing <html>
```

```
element
console.log(window['jQuery'] === undefined); //logs true

//Only after everything is loaded can we safley conclude that jQuery was loaded and p
arsed
document.body.onload = function(){console.log(jQuery().jquery)}; //logs function
</script>

</body>
</html>
```

Notes

According to the specification defered scripts are suppose to be exectued in document order and before the `DOMContentLoaded` event. However, adherence to this specification among modern browsers is inconsistent.

`defer` is a boolan attribute it does not have a value

Some browers support defered inline scripts but this is not common among modern browsers

By using `defer` the assummption is that `document.write()` is not being used in the JavaScript that will be defered

10.4 Asynchronously downloading & executing external JavaScript files using `async`

The `<script>` element has an attribute called `async` that will override the sequential blocking nature of `<script>` elements when the DOM is being constructed by a web browser. By using this attribute, we are telling the browser not to block the construction (i.e. DOM parsing, including downloading other assets e.g. images, style sheets, etc...) of the html page and forgo the the sequential loading as well.

What happens by using the `async` attribute is the files are loaded in parallel and parsed in order of download once they are fully downloaded. In the code below I comment what is happening when the HTML document is being parsed and render by the web browser.

live code: http://jsfiddle.net/domenlightenment/

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- Don't block, just start downloading and then parse the file when it's done downl
oading -->
<script async src="http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.3/undersco
re-min.js"></script>
```

```
<!-- Don't block, just start downloading and then parse the file when it's done downl
oading -->
<script async src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.min.js">
</script>

<!-- Don't block, just start downloading and then parse the file when it's done downl
oading -->
<script async src="http://cdnjs.cloudflare.com/ajax/libs/jquery-mousewheel/3.0.6/jque
ry.mousewheel.min.js"></script>


<script>
// we have no idea if jQuery has been loaded yet likley not yet...
console.log(window['jQuery'] === undefined);//logs true

//Only after everything is loaded can we safley conclude that jQuery was loaded and p
arsed
document.body.onload = function(){console.log(jQuery().jquery)};
</script>

</body>
</html>
```

Notes

IE 10 has support for `async`, but IE 9 does not

A major drawback to using the `async` attribute is JavaScript files potentially get parsed out of the order they are included in the DOM. This raises a dependency management issue.

`async` is a boolan attribute it does not have a value

By using `async` the assummption is that `document.write()` is not being used in the JavaScript that will be defered

The `async` attribute will trump the `defer` if both are used on a `<script>` element

10.5 Forcing asynchronous downloading & parsing of external JavaScript using dynamic `<script>`

A known hack for forcing a web browser into asynchronous JavaScript downloading and parsing without using the `async` attribure is to programatically create `<script>` elements that include external JavaScript files and insert them in the DOM. In the code below I programatically create the `<script>` element node and then append it to the `<body>` element which forces the browser to treat the `<script>` element asynchronously.

live code: http://jsfiddle.net/domenlightenment/du94d

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- Don't block, just start downloading and then parse the file when it's done downl
oading -->
<script>
var underscoreScript = document.createElement("script");
underscoreScript.src = "http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.3/und
erscore-min.js";
document.body.appendChild(underscoreScript);
</script>

<!-- Don't block, just start downloading and then parse the file when it's done downl
oading -->
<script>
var jqueryScript = document.createElement("script");
jqueryScript.src = "http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.min.js"
;
document.body.appendChild(jqueryScript);
</script>

<!-- Don't block, just start downloading and then parse the file when it's done downl
oading -->
<script>
var mouseWheelScript = document.createElement("script");
mouseWheelScript.src = "http://cdnjs.cloudflare.com/ajax/libs/jquery-mousewheel/3.0.6
/jquery.mousewheel.min.js";
document.body.appendChild(mouseWheelScript);
</script>

<script>
//Only after everything is loaded can we safley conclude that jQuery was loaded and p
arsed
document.body.onload = function(){console.log(jQuery().jquery)};
</script>

</body>
</html>
```

Notes

A major drawback to using dynamic `<script>` elements is JavaScript files potentially get parsed out of the order they are included in the DOM. This raises a dependency management issue.

10.6 Using the `onload` call back for asynchronous `<script>`'s so we know when its loaded

The `<script>` element supports a load event handler (ie. `onload`) that will execute once an external JavaScript file has been loaded and executed. In the code below I leverage the `onload` event to create a callback programatically notifying us when the JavaScript file has been downloaded and exectued.

live code: http://jsfiddle.net/domenlightenment/XzAFx

```
<!DOCTYPE html>
<html lang="en">
<body>

<!-- Don't block, just start downloading and then parse the file when it's done downl
oading -->
<script>
var underscoreScript = document.createElement("script");
underscoreScript.src = "http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.3/und
erscore-min.js";
underscoreScript.onload = function(){console.log('underscsore is loaded and exectuted
');};
document.body.appendChild(underscoreScript);
</script>

<!-- Don't block, just start downloading and then parse the file when it's done downl
oading -->
<script async src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.min.js"
onload="console.log('jQuery is loaded and exectuted');"></script>

</body>
</html>
```

Notes

The `onload` event is only the tip of the iceberg avaliable where `onload` is supported you also have use of `onerror`, `load`, and, `error`.

10.7 Be mindful of `<script>`'s placement in HTML for DOM manipulation

Given a `<script>` elements synchronous nature, placing one in the `<head>` element of an HTML document presents a timing problem if the JavaScript execution is dependant upon any of the DOM that proceeds the `<script>`. In a nut shell, if JavaScript is executed at the begining of a document that manipulates the DOM, that proceeds it, you are going to get a JavaScript error. Proven by the following code example:

live code: N/A

```
<!DOCTYPE html>
<html lang="en">
<head>
<!-- stop parsing, block parsing, exectue js then resume... -->
<script>
//we can't script the body element yet, its null, not even been parsed by the browser
, its not in the DOM yet
console.log(document.body.innerHTML); //logs Uncaught TypeError: Cannot read property
 'innerHTML' of null
</script>
</head>
<body>
<strong>Hi</strong>
</body>
</html>
```

Many developers, myself being one of them, for this reason will attempt to place all `<script>` elements before the closing `</body>` element. By doing this you can rest assured the DOM in front of the `<script>`'s has been parsed and is ready for scripting. As well, this strategy will remove a dependancy on DOM ready events that can liter a code base.

10.8 Getting a list of `<script>`'s in the DOM

The `document.scripts` property avaliable from the document object provides a list (i.e. an `HTMLCollection`) of all of the scripts currently in the DOM. In the code below I leverage this property to gain access to each of the `<script>` elements `src` attributes.

live code: N/A

```
<!DOCTYPE html>
<html lang="en">
<body>
<script src="http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.3/underscore-min
.js"></script>
<script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.min.js"></scri
pt>
<script src="http://cdnjs.cloudflare.com/ajax/libs/jquery-mousewheel/3.0.6/jquery.mou
sewheel.min.js"></script>

<script>
Array.prototype.slice.call(document.scripts).forEach(function(elm){
        console.log(elm);
});//will log each script element in the document
</script>

</body>
</html>
```

Chapter 11 - DOM Events

11.1 DOM events overview

An event, in terms of the DOM, is either a pre-defined or custom moment in time that occurs in relationship with an element in the DOM, the `document` object, or the `window` object. These moments are typically predetermined and programaticlly accounted for by associating functionality (i.e. handlers/callbacks) to occur when these moments in time come to pass. These moments can be initiated by that state of the UI (e.g. input is focused or something has been dragged), the state of the enviroment that is running the JavaScript program (e.g. page is loaded or XHR request has finished), or the state of the program itself (e.g. start monitor users ui interaction for 30 seconds after the page has loaded).

Setting up events can be accomplished using inline attribute event handlers, property event handlers, or the `addEventListener()` method. In the code below I'm demonstrating these three patterns for setting up an event. All three patterns add a `click` event that is invoked whenever the `<div>` in the html document is clicked by the mouse.

live code: http://jsfiddle.net/domenlightenment/4EPjN

```
<!DOCTYPE html>
<html lang="en">

<!-- inline attribure event handler pattern -->
<body onclick="console.log('fire/trigger attribure event handler')">

<div>click me</div>

<script>
var elementDiv = document.querySelector('div');

// property event handler pattern
elementDiv.onclick = function(){console.log('fire/trigger property event handler')};

//addEventListener method pattern
elementDiv.addEventListener('click',function(){console.log('fire/trigger addEventList
ener')}, false);
```

```
</script>
</body>
</html>
```

Notice that one of the events is attached to the `<body>` element. If you find it odd that the attribute event handler on the `<body>` fires by clicking the `<div>` element consider that when the `<div>` is clicked, are you not also clicking on the `<body>` element. Click anywhere but on the `<div>` and you still see the attribute handler fire on the `<body>` element alone.

While all three of these patterns for attaching an event to the DOM programatically schedule the event, only the `addEventListener()` provides a robust and organized solution. The inline attribute event handler mixes together JavaScript and HTML and best practices advise keeping these things seperate.

The downside to using a property event handler is that only one value can be assigned to the event property at a time. Meaning, you can't add more than one propety event handler to a DOM node when assigning events as property values. The code below shows an example of this by assigning a value to the `onclick` property twice, the last value set is used when the event is invoked.

live code: http://jsfiddle.net/domenlightenment/U8bWR

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>
var elementDiv = document.querySelector('div');

// property event handler
elementDiv.onclick = function(){console.log('I\'m first, but I get overidden/replace'
)};

//overrides/replaces the prior value
elementDiv.onclick = function(){console.log('I win')};

</script>
</body>
</html>
```

Additionaly, using event handlers inline or property event handlers can suffer from scoping nuances as one attempts to leverage the scope chain from the function that is invoked by the event. The `addEventListener()` smooths out all of these issues, and will be used throughout this chapter.

Notes

Element nodes typically support inline event handlers (e.g. *<div onclick=""></div>*), property event handlers (e.g. *document.querySelector('div').onclick = function(){}*), and the use of the *addEventListener()* method.

The *Document* node supports property event handlers (e.g. *document.onclick = funciton()*) and the use of the *addEventListener()* method.

The *window* object supports inline event handler's via the *<body>* or *<frameset>* element (e.g. *<body onload=""></body>*), property event handlers (e.g. *window.load = function(){}*), and the use of the *addEventListener()* method.

A property event handler historically has been refered to as a "DOM level 0 event". And the *addEventListener()* is often refered to as a "DOM level 2 event". Which is rather confusing considering there is no level 0 event or level 1 event. Addintioanlly, inline event handlers are known to be called, "HTML event handlers".

11.2 DOM event types

In the tables below I detail the most common pre-defined events that can be attached to *Element* nodes, the *document* object, and the *window* object. Of course not all events are directly applicable to the node or object it can be attached too. That is, just because you can attach the event without error, and most likley invoke the event (i.e. bubbling events like *onchange* to *window*), does not mean that adding something like *window.onchange* is logical given that this event, by design was not meant for the *window* object.

User interface events

Event Type	Event Interface	Description	Event Targets	Bubbles	Cancelable
load	*Event*, *UIEvent*	fires when an asset (HTML page, image, CSS, frameset, *<object>*, or JS file) is loaded.	*Element*, *Document*, *window*, *XMLHttpRequest*, *XMLHttpRequestUpload*	No	No
unload	*UIEvent*	fires when user agent removes the resource (document, element, defaultView) or any depending resources (images, CSS file, etc.)	*window*, *<body>*, *<frameset>*	No	No
abort	*Event*, *UIEvent*	Fires when an resource (object/image) is	*Element*, *XMLHttpRequest*,	Yes	No

		stopped from loading before completely loaded	*XMLHttpRequestUpload*		
error	*Event*, *UIEvent*	Fires when a resource failed to load, or has been loaded but cannot be interpreted according to its semantics, such as an invalid image, a script execution error, or non-well-formed XML	*Element*, *XMLHttpRequest*, *XMLHttpRequestUpload*	Yes	No
resize	*UIEvent*	Fires when a document view has been resized. This event type is dispatched after all effects for that occurrence of resizing of that particular event target have been executed by the user agent	*window*, *<body>*, *<frameset>*	Yes	No
scroll	*UIEvent*	Fires when a user scrolls a document or an element.	*Element*, *Document*, *window*	Yes	No
contextmenu	*MouseEvent*	fires by right clicking an element	*Element*	Yes	Yes

Focus events

Event Type	Event Interface	Description	Events Targets	Bubbles	Cancelable
blur	*FocusEvent*	Fires when an element loses focus either via the mouse or tabbing	*Element* (except *<body>* and *<frameseet>*), *Document*	No	No
focus	*FocusEvent*	Fires when an element receives focus	*Element* (except *<body>* and *<frameseet>*), *Document*	No	No
focusin	*FocusEvent*	Fires when an event target is about to receive focus but before the focus is shifted. This event occurs right before the focus event	*Element*	Yes	No
focusout	*FocusEvent*	Fires when an event target is	*Element*	Yes	No

		about to lose focus but before the focus is shifted. This event occurs right before the blur event			

Form events

Event Type	Event Interface	Description	Event Targets	Bubbles	Cancelable
change	specific to HTML forms	Fires when a control loses the input focus and its value has been modified since gaining focus	*Element*	Yes	No
reset	specific to HTML forms	Fires when a form is reset	*Element*	Yes	No
submit	specific to HTML forms	Fires when a form is submitted	*Element*	Yes	Yes
select	specific to HTML forms	Fires when a user selects some text in a text field, including input and textarea	*Element*	Yes	No

Mouse events

Event Type	Event Interface	Description	Event Targets	Bubbles	Cancelable
click	*MouseEvent*	Fires when mouse pointer is clicked (or user presses enter key) over an element. A click is defined as a mousedown and mouseup over the same screen location. The sequence of these events is *mousedown*>*mouseup*>*click*. Depending upon the environment configuration, the click event may be dispatched if one or more of the event types mouseover, mousemove, and mouseout occur between the press and release of the pointing device button. The click event may also be followed by the dblclick event	*Element*, *Document*, *window*	Yes	Yes
dblclick	*MouseEvent*	Fires when a mouse pointer is clicked twice over an element. The definition of a double click depends on the environment configuration, except that	*Element*, *Document*, *window*	Yes	Yes

		the event target must be the same between *mousedown*, *mouseup*, and *dblclick*. This event type must be dispatched after the event typeclick if a click and double click occur simultaneously, and after the event type *mouseup* otherwise			
mousedown	*MouseEvent*	Fires when mouse pointer is pressed over an element	*Element*, *Document*, *window*	Yes	Yes
mouseenter	*MouseEvent*	Fires when mouse pointer is moved onto the boundaries of an element or one of its descendent elements. This event type is similar to mouseover, but differs in that it does not bubble, and must not be dispatched when the pointer device moves from an element onto the boundaries of one of its descendent elements	*Element*, *Document*, *window*	No	No
mouseleave	*MouseEvent*	Fires when mouse pointer is moved off of the boundaries of an element and all of its descendent elements. This event type is similar to mouseout, but differs in that does not bubble, and that it must not be dispatched until the pointing device has left the boundaries of the element and the boundaries of all of its children	*Element*, *Document*, *window*	No	No
mousemove	*MouseEvent*	Fires when mouse pointer is moved while it is over an element. The frequency rate of events while the pointing device is moved is implementation-, device-, and platform-specific, but multiple consecutive mousemove events should be fired for sustained pointer-device movement, rather than a single event for each instance of mouse movement. Implementations are encouraged to determine the optimal frequency rate to balance responsiveness with performance	*Element*, *Document*, *window*	Yes	No
mouseout	*MouseEvent*	Fires when mouse pointer is moved off of the boundaries of an element. This event type is similar to *mouseleave*, but differs in that does bubble, and that it must be dispatched when the pointer device moves from an element onto the boundaries of one of its descendent elements	*Element*, *Document*, *window*	Yes	Yes

mouseup	*MouseEvent*	Fires when mouse pointer button is released over an element	*Element*, *Document*, *window*	Yes	Yes

Wheel events

Event Type	Event Interface	Description	Event Targets	Bubbles	Cancelable
wheel (browsers use *mousewheel* but the specification uses *wheel*)	*WheelEvent*	Fires when a mouse wheel has been rotated around any axis, or when an equivalent input device (such as a mouse-ball, certain tablets or touchpads, etc.) has emulated such an action. Depending on the platform and input device, diagonal wheel deltas may be delivered either as a singlewheel event with multiple non-zero axes or as separate wheel events for each non-zero axis. Some helpful details about browser support can be found here.	*Element*, *Document*, *Window*	Yes	Yes

Keyboard events

Event Type	Event Interface	Description	Event Targets	Bubbles	Cancelable
keydown	*KeyboardEvent*	Fires when a key is initially pressed. This is sent after any key mapping is performed, but before any input method editors receive the keypress. This is sent for any key, even if it doesn't generate a character code.	*Element*, *Document*	Yes	Yes
keypress	*KeyboardEvent*	Fires when a key is initially pressed, but only if that key normally produces a character value. This is sent after any key mapping is performed, but before any input method editors receive the keypress.	*Element*, *Document*	Yes	Yes
keyup	*KeyboardEvent*	Fires when a key is released. This is sent after any key mapping is performed, and always follows thecorresponding *keydown* and *keypress* events.	*Element*, *Document*	Yes	Yes

Touch events

Event Type	Event Interface	Description	Event Targets	Bubbles	Cancelable
touchstart	*TouchEvent*	Fires event to indicate when the user places a touch point on the touch surface	*Element*, *Document*, *window*	Yes	Yes
touchend	*TouchEvent*	Fires event to indicate when the user removes a touch point from the touch surface, also including cases where the touch point physically leaves the touch surface, such as being dragged off of the screen	*Element*, *Document*, *window*	Yes	Yes
touchmove	*TouchEvent*	Fires event to indicate when the user moves a touch point along the touch surface	*Element*, *Document*, *window*	Yes	Yes
touchenter	*TouchEvent*	Fires event to indicate when a touch point moves onto the interactive area defined by a DOM element	*Element*, *Document*, *window*	No	?
toucheleave	*TouchEvent*	Fires event to indicate when a touch point moves off the interactive area defined by a DOM element	*Element*, *Document*, *window*	No	?
touchcancel	*TouchEvent*	Fires event to indicate when a touch point has been disrupted in an implementation-specific manner, such as a synchronous event or action originating from the UA canceling the touch, or the touch point leaving the document window into a non-document area which is capable of handling user interactions.	*Element*, *Document*, *window*	Yes	No

Notes

Touch events are typically only supported iOS, Andorid, and Blackberry browsers or browsers (e.g. chrome) that can switch on touch modes

Window, *<body>*, and frame specific events

Event Type	Event Interface	Description	Event Targets	Bubbles	Cancelable
afterprint	?	Fires on the object immediately after its	*window*, *<body>*,	No	No

		associated document prints or previews for printing	`<frameset>`		
beforeprint	?	Fires on the object before its associated document prints or previews for printing	*window*, *<body>*, *<frameset>*	No	No
beforeunload	?	Fires prior to a document being unloaded	*window*, *<body>*, *<frameset>*	No	Yes
hashchange	*HashChangeEvent*	Fires when there are changes to the portion of a URL that follows the number sign (#)	*window*, *<body>*, *<frameset>*	No	No
messsage	?	Fires when the user sends a cross-document message or a message is sent from a *Worker* with *postMessage*	*window*, *<body>*, *<frameset>*	No	No
offline	*NavigatorOnLine*	Fires when browser is working offline	*window*, *<body>*, *<frameset>*	No	No
online	*NavigatorOnLine*	Fires when browser is working online	*window*, *<body>*, *<frameset>*	No	No
pagehide	*PageTransitionEvent*	Fires when traversing from a session history entry	*window*, *<body>*, *<frameset>*	No	No
pageshow	*PageTransitionEvent*	The pagehide event is fired when traversing from a session history entry	*window*, *<body>*, *<frameset>*	No	No

Document specific events

Event Type	Event Interface	Description	Event Targets	Bubbles	Cancelable
readystatechange	*Event*	Fires event when *readyState* is changed	*Document*, *XMLHttpRequest*	No	No
DOMContentLoaded	*Event*	Fires when a webpage has been parsed, but before all resources have been fully downloaded	*Document*	Yes	No

Drag events

Event Type	Event Interface	Description	Event Targets	Bubbles	Cancelable
`drag`	`DragEvent`	Fires on the source object continuously during a drag operation.	`Element`, `Document`, `window`	Yes	Yes
`dragstart`	`DragEvent`	Fires on the source object when the user starts to drag a text selection or selected object. The ondragstart event is the first to fire when the user starts to drag the mouse.	`Element`, `Document`, `window`	Yes	Yes
`dragend`	`DragEvent`	Fires on the source object when the user releases the mouse at the close of a drag operation. The ondragend event is the final drag event to fire, following the ondragleave event, which fires on the target object.	`Element`, `Document`, `window`	Yes	No
`dragenter`	`DragEvent`	Fires on the target element when the user drags the object to a valid drop target.	`Element`, `Document`, `window`	Yes	Yes
`dragleave`	`DragEvent`	Fires on the target object when the user moves the mouse out of a valid drop target during a drag operation.	`Element`, `Document`, `window`	Yes	No
`dragover`	`DragEvent`	Fires on the target element continuously while the user drags the object over a valid drop target. The ondragover event fires on the target object after the ondragenter event has fired.	`Element`, `Document`, `window`	Yes	Yes
`drop`	`DragEvent`	Fires on the target object when the mouse button is released during a drag-and-drop operation. The ondrop event fires before the ondragleave and ondragend events.	`Element`, `Document`, `window`	Yes	Yes

Notes

The tables below were crafted from the following three resources Document Object Model (DOM) Level 3 Events Specification 5 User Event Module, DOM event reference, HTML Living Standard 7.1.6 Event handlers on elements, Document objects, and Window objects,, and Event compatibility tables.

I've only mentioned here in this section the most common event types. Keep in mind there are numerous HTML5 api's that I've excluded from the this section (e.g. media events for `<video>` and `<audio>` elements or all state change events for the XMLHttpRequest Level 2).

The `copy`, `cut`, and `textinput` event are not defined by DOM 3 events or HTML5

11.3 The event flow

When an event is invoked the event flows or propagates through the DOM, firing the same event on other nodes and JavaScript objects. The event flow can be programmed to occur as a capture phase (i.e. DOM tree trunk to branch) or bubbling phase (i.e. DOM tree branches to trunk), or both.

In the code below I set up 10 event listeners that can all be invoked, due to the event flow, by clicking once on the `<div>` element in the HTML document. When the `<div>` is clicked the capture phase begins at the `window` object and propagates down the DOM tree firing the `click` event for each object (i.e. `window` **>** `document` **>** `<html>` **>** `<body>` **>** event target) until it hits the event target. Once the capture phase ends the target phase starts, firing the `click` event on the target element itself. Next the propagation phase propagates up from the event target firing the `click` event until it reaches the `window` object (i.e. event target **>** `<body>` **>** `<html>` **>** `document` **>** `window`). With this knowledge it should be obvious why clicking the `<div>` in the code example logs to the console 1,2,3,4,5,6,7,8,9,11.

live code: http://jsfiddle.net/domenlightenment/CAdTv

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me to start event flow</div>

<script>

/*notice that I am passing the addEventListener() a boolean parameter of true so capt
ure events fire, not just bubbling events*/

//1 capture phase
window.addEventListener('click',function(){console.log(1);},true);

//2 capture phase
document.addEventListener('click',function(){console.log(2);},true);

//3 capture phase
document.documentElement.addEventListener('click',function(){console.log(3);},true);

//4 capture phase
document.body.addEventListener('click',function(){console.log(4);},true);

//5 target phase occurs during capture phase
document.querySelector('div').addEventListener('click',function(){console.log(5);},tr
ue);
```

```
//6 target phase occurs during bubbling phase
document.querySelector('div').addEventListener('click',function(){console.log(6);},fa
lse);

//7 bubbling phase
document.body.addEventListener('click',function(){console.log(7);},false);

//8 bubbling phase
document.documentElement.addEventListener('click',function(){console.log(8);},false);

//9 bubbling phase
document.addEventListener('click',function(){console.log(9);},false);

//10 bubbling phase
window.addEventListener('click',function(){console.log(10)},false);

</script>
</body>
</html>
```

After the `<div>` is clicked, the event flow proceeds in this order:

1. capture phase invokes click events on window that are set to fire on capture
2. capture phase invokes click events on document that are set to fire on capture
3. capture phase invokes click events on html element that are set to fire on capture
4. capture phase invokes click events on body element that are set to fire on capture
5. target phase invokes click events on div element that are set to fire on capture
6. target phase invokes click events on div element that are set to fire on bubble
7. bubbling phase invokes click events on body element are set to fire on bubble
8. bubbling phase invokes click events on html element are set to fire on bubble
9. bubbling phase invokes click events on document are set to fire on bubble
10. bubbling phase invokes click events on window are set to fire on bubble

The use of the capture phase is not all that common due to a lack of browser support for this phase. Typically events are assumed to be inovked during the bubbling phase. In the code below I remove the capture phase from the previous code example and demostrate typically what is occuring during an event invocation.

live code: http://jsfiddle.net/domenlightenment/C6qmZ

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me to start event flow</div>
```

```
<script>

//1 target phase occurs during bubbling phase
document.querySelector('div').addEventListener('click',function(){console.log(1);},fa
lse);

//2 bubbling phase
document.body.addEventListener('click',function(){console.log(2);},false);

//3 bubbling phase
document.documentElement.addEventListener('click',function(){console.log(3);},false);

//4 bubbling phase
document.addEventListener('click',function(){console.log(4);},false);

//5 bubbling phase
window.addEventListener('click',function(){console.log(5)},false);

</script>
</body>
</html>
```

Notice in the last code example that if the click event is initiated (click anywhere except on the `<div>`) on the `<body>` element the click event attached to the `<div>` is not invoked and bubbling invocation starts on the `<body>`. This is due to the fact the the event target is no longer the `<div>` but instead the `<body>` element.

Notes

Modern browsers do support the use of the capture phase so what was once considered unreliable might just server some value today. For example, one could intercept an event before it occurs on the event target.

Keep this knowledge of event capturing and bubbling at the forefront of your thoughts when you read the event delegation section of this chapter.

The event object passed to event listener functions contains a `eventPhase` property containing a number which indicates which phase an event is inoked in. A value of 1 indicates the capture phase. A value of 2 indicates the target phase. And a value of 3 indicates bubbling phase.

11.4 Adding event listeners to `Element` nodes, `window` object, and `Document` object

The `addEventListener()` method is avaliabe on all `Element` nodes, the `window` object, and the `document` object providing the ability to added event listeners to parts of an HTML document as well as JavaScript objects relating to the DOM and BOM (browser object model). In the code below I leverage this method to add a `mousemove` event to a `<div>` element, the `document` object, and the `window` object. Notice, due to the event flow, that mouse movement specifically over the `<div>` will invoke all three listeners each to time a movement occurs.

live code: http://jsfiddle.net/domenlightenment/sSFK5

```
<!DOCTYPE html>
<html lang="en">

<body>

<div>mouse over me</div>

<script>

//add a mousemove event to the window object, invoking the event during the bubbling
phase
window.addEventListener('mousemove',function(){console.log('moving over window');},fa
lse);

//add a mousemove event to the document object, invoking the event during the bubblin
g phase
document.addEventListener('mousemove',function(){console.log('moving over document');
},false);

//add a mousemove event to a <div> element object, invoking the event during the bubb
ling phase
document.querySelector('div').addEventListener('mousemove',function(){console.log('mo
ving over div');},false);

</script>
</body>
</html>
```

The `addEventListener()` method used in the above code example takes three arguments. The first argument is the type of event to listen for. Notice that the event type string does not contain the "on" prefix (i.e. `onmousemove`) that event handlers require. The second argument is the function to be invoked when the event occurs. The third parameter is a boolean indicating if the event should be fired during the capture phase or bubbling phase of the event flow.

Notes

I've purposfully avoided dicussing inline event handlers & property event handlers in favor of promoting the use of `addEventListener()`

Typically a developer wants events to fire during the bubbling phase so that object eventing handles the event before bubbling the event up the DOM. Because of this you almost always provide a `false` value as the last argument to the `addEventListener()`. In modern browsers if the 3rd parameter is not specified it will default to false.

You should be aware that the `addEventListener()` method can be used on the `XMLHttpRequest` object

11.5 Removing event listeners

The `removeEventListener()` method can be used to remove events listeners, if the orginal listener was not added using an anonymous function. In the code below I add two events listeners to the HTML document and attempt to remove both of them. However, only the listener that was attached using a function reference is removed.

live code: http://jsfiddle.net/domenlightenment/XP2Ug

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click to say hi</div>

<script>

var sayHi = function(){console.log('hi')};

//adding event listener using anonymous function
document.body.addEventListener('click',function(){console.log('dude');},false);

//adding event listener using function reference
document.querySelector('div').addEventListener('click',sayHi,false);

//attempt to remove both event listeners, but only the listener added with a funtions
 reference is removed
document.querySelector('div').removeEventListener('click',sayHi,false);

//this of course does not work as the function passed to removeEventListener is a new
 and different function
document.body.removeEventListener('click',function(){console.log('dude');},false);

//clicking the div will still invoke the click event attached to the body element, th
is event was not removed

</script>
</body>
</html>
```

Anonymous functions added using `addEventListener()` method simply cannot be removed.

11.6 Getting event properties from the event object

The handler or callback function invoked for events is sent by default a parameter that contains all relevant information about an event itself. In the code below I demostrate access to this event object and log all of its properties and values for a load event as well as a click event. Make sure you click the `<div>` to see the properties assocaited with a click event.

live code: http://jsfiddle.net/domenlightenment/d4SnQ

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

document.querySelector('div').addEventListener('click',function(event){
Object.keys(event).sort().forEach(function(item){
     console.log(item+' = '+event[item]); //logs event propeties and values
});
},false);

//assumes 'this' is window
this.addEventListener('load',function(event){
Object.keys(event).sort().forEach(function(item){
     console.log(item+' = '+event[item]); //logs event propeties and values
});
},false);

</script>
</body>
</html>
```

Keep in mind that each event will contain slightly different properties based on the event type (e.g. MouseEvent, KeyboardEvent, WheelEvent).

Notes

The event object also provides the `stopPropagation()`, `stopImediatePropagation()`, and `preventDefault()` methods.

In this book I use the argument name `event` to reference the event object. In truth you can use any name you like and its not uncommon to see `e` or `evt`.

11.7 The value of `this` when using `addEventListener()`

The value of `this` inside of the event listener function passed to the `addEventListener()` method will be a reference to the node or object the event is attached too. In the code below I attach an event to a `<div>` and then using `this` inside of the event listener gain access to the `<div>` element the event is attached too.

live code: http://jsfiddle.net/domenlightenment/HwKgH

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

document.querySelector('div').addEventListener('click',function(){
// 'this' will be the element or node the event listener is attached too
console.log(this); //logs '<div>'
},false);

</script>
</body>
</html>
```

When events are invoked as part of the event flow the `this` value will remain the value of the node or object that the event listener is attached too. In the code below we add a `click` event listener to the `<body>` and regardless of if you click on the `<div>` or the `<body>` the value of `this` always points to `<body>`.

live code: http://jsfiddle.net/domenlightenment/NF2gn

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

//click on the <div> or the <body> the value of this remains the <body> element node
document.body.addEventListener('click',function(){
console.log(this); //log <body>...</body>
},false);

</script>
</body>
</html>
```

Additionally its possible using the `event.currentTarget` property to get the same reference, to the node or object invoking the event listener, that the `this` property provides. In the code below I leverage the `event.currentTarget` event object property showcasing that it returns the same value as `this`.

live code: http://jsfiddle.net/domenlightenment/uQm3f

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

document.addEventListener('click',function(event){
console.log(event.currentTarget);  //logs '#document'
//same as...
console.log(this);
},false);

document.body.addEventListener('click',function(event){
console.log(event.currentTarget); //logs '<body>'
//same as...
console.log(this);
},false);

document.querySelector('div').addEventListener('click',function(event){
console.log(event.currentTarget); //logs '<div>'
//same as...
console.log(this);
},false);

</script>
</body>
</html>
```

11.8 Referencing the `target` of an event and not the node or object the event is invoked on

Because of the event flow its possible to click a `<div>`, contained inside of a `<body>` element and have a `click` event listener attached to the `<body>` element get invoked. When this happens, the event object passed to the event listener function attached to the `<body>` provides a reference (i.e. `event.target`) to the node or object that the event originated on (i.e. the target). In the code below when the `<div>` is clicked, the `<body>` element's `click` event listener is invoked and the `event.target` property references the orginal `<div>` that was the target of the click event. The `event.target` can be extremely useful when an event that fires because of the event flow needs knowledge about the origin of the event.

live code: http://jsfiddle.net/domenlightenment/dGkTQ

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

document.body.addEventListener('click',function(event){
//when the <div> is clicked logs '<div>' because the <div> was the target in the even
t flow
console.log(event.target);
},false);

</script>
</body>
</html>
```

Consider that in our code example if the `<body>` element is clicked instead of the `<div>` then the event `target` and the element node that the event listener is invoked on are the same. Therefore `event.target`, `this`, and `event.currentTarget` will all contain a reference to the `<body>` element.

11.9 Cancelling default browser events using `preventDefault()`

Browsers provide several events already wired up when an HTML page is presented to a user. For example, clicking a link has a corresponding event (i.e. you navigate to a url). So does clicking a checkbox (i.e. box is checked) or typing text into a text field (i.e. text is inputed and appears on screen). These browser events can be prevented by calling the `preventDefault()` method inside of the event handler function associated with a node or object that invokes a browser default event. In the code below I prevent the default event that occurs on a `<a>`, `<input>`, and `<textarea>`.

live code: http://jsfiddle.net/domenlightenment/Ywcyh

```
<!DOCTYPE html>
<html lang="en">
<body>
```

```
<a href="google.com">no go</div>

<input type="checkbox" />

<textarea></textarea>

<script>

document.querySelector('a').addEventListener('click',function(event){
event.preventDefault(); //stop the default event for <a> which would be to load a url
},false);

document.querySelector('input').addEventListener('click',function(event){
event.preventDefault(); //stop default event for checkbox, which would be to toggle c
heckbox state
},false);

document.querySelector('textarea').addEventListener('keypress',function(event){
event.preventDefault(); //stop default event for textarea, which would be to add char
acters typed
},false);

/*keep in mind that events still propagate, clicking the link in this html document w
ill stop the default event but not event bubbling*/
document.body.addEventListener('click',function(){
console.log('the event flow still flows!');
},false);

</script>
</body>
</html>
```

All attempts to click the link, check the box, or type in the text input in the previous code example will fail because I am preventing the default events for these elements from occuring.

Notes

The `preventDefault()` methods does not stop events from propagating (i.e. bubbling or capture phases)

Providing a `return false` at the end of the body of the event listener has the same result as call the `preventDefault()` method

The event object passed to event listener functions contains a boolean `cancelable` property which indicates if the event will respond to preveentDefault() method and canceling default behavior

The event object passed to event listener functions contains a `defaultPrevented` property which indicates true if `preventDefault()` has been invoked for a bubbling event.

11.10 Stoping the event flow using `stopPropagation()`

Calling `stopProgagation()` from within an event handler/listener will stop the capture and bubble event flow phases, but any events directly attached to the node or object will still be invoked. In the code below the `onclick` event attached to the `<body>` is never gets invoked because we are stopping the event from bubbling up the DOM when clicking on the `<div>`.

live code: http://jsfiddle.net/domenlightenment/RFKmA

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

document.querySelector('div').addEventListener('click',function(){
console.log('me too, but nothing from the event flow!');
},false);

document.querySelector('div').addEventListener('click',function(event){
console.log('invoked all click events attached, but cancel capture and bubble event p
hases');
event.stopPropagation();
},false);

document.querySelector('div').addEventListener('click',function(){
console.log('me too, but nothing from the event flow!');
},false);

/*when the <div> is clicked this event is not invoked because one of the events attac
hed to the <div> stops the capture and bubble flow.*/
document.body.addEventListener('click',function(){
console.log('What, denied from being invoked!');
},false);

</script>
</body>
</html>
```

Notice that other click events attached to the the `<div>` still get invoked! Additionally using `stopPropagation()` does not prevent default events. Had the `<div>` in our code example been a `<a>` with an href value calling stopPropagation would not have stopped the browser default events from getting invoked.

11.11 Stoping the event flow as well as other like events on the same target using `stopImmediatePropagation()`

Calling the `stopImmediatePropagation()` from within an event handler/listener will stop the event flow phases (i.e. `stopPropagation()`), as well as any other like events attached to the event target that are attached after the event listener that invokes the `stopImmediatePropagation()` method. In the code example below If we call `stopImmediatePropagation()` from the second event listener attached to the `<div>` the click event that follows will not get invoked.

live code: http://jsfiddle.net/domenlightenment/znSjM

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

//first event attached
document.querySelector('div').addEventListener('click',function(){
console.log('I get invoked because I was attached first');
},false);

//seond event attached
document.querySelector('div').addEventListener('click',function(event){
console.log('I get invoked, but stop any other click events on this target');
event.stopImmediatePropagation();
},false);

//third event attached, but because stopImmediatePropagation() was called above this
event does not get invoked
document.querySelector('div').addEventListener('click',function(){
console.log('I get stopped from the previous click event listener');
},false);

//notice that the event flow is also cancelled as if stopPropagation was called too
document.body.addEventListener('click',function(){
console.log('What, denied from being invoked!');
},false);

</script>
</body>
</html>
```

Notes

Using the `stopImmediatePropagation()` does not prevent default events. Browser default events still get invoked and only calling `preventDefault()` will stop these events.

11.12 Custom events

A developer is not limited to the predefined event types. Its possible to attach and invoke a custom event, using the `addEventListener()` method like normal in combiniation with `document.createEvent()`, `initCustomEvent()`, and `dispatchEvent()`. In the code below I create a custom event called `goBigBlue` and invoke that event.

live code: http://jsfiddle.net/domenlightenment/fRndj

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>click me</div>

<script>

var divElement = document.querySelector('div');

//create the custom event
var cheer = document.createEvent('CustomEvent'); //the 'CustomEvent' parameter is req
uired

//create an event listener for the custom event
divElement.addEventListener('goBigBlue',function(event){
    console.log(event.detail.goBigBlueIs)
},false);

/*Use the initCustomEvent method to setup the details of the custom event.
Parameters for initCustomEvent are: (event, bubble?, cancelable?, pass values to even
t.detail)*/
cheer.initCustomEvent('goBigBlue',true,false,{goBigBlueIs:'its gone!'});

//invoke the custom event using dispatchEvent
divElement.dispatchEvent(cheer);

</script>
</body>
</html>
```

Notes

IE9 requires (not optinal) the fourth parameter on `initiCustomEvent()`

The DOM 4 specifiction added a `CustomEvent()` constructor that has simplified the life cycle of a custom event but its not supported in ie9 and as of this writting and is still in flux

11.13 Simulating/Triggering mouse events

Simiulating an event is not unlike creating a custom event. In the case of simulating a mouse event we create a `'MouseEvent'` using `document.createEvent()`. Then, using `initMouseEvent()` we setup the mouse event that is going to occur. Next the mouse event is dispatched on the element that we'd like to simulate an event on (i.e the `<div>` in the html document). In the code below a click event is attached to the `<div>` in the page. Instead of clicking the `<div>` to invoke the click event the event is triggered or simulated by programatically setting up a mouse event and dispatching the event to the `<div>`.

live code: http://jsfiddle.net/domenlightenment/kx7zJ

```
<!DOCTYPE html>
<html lang="en">
<body>

<div>no need to click, we programatically trigger it</div>

<script>

var divElement = document.querySelector('div');

//setup click event that will be simulated
divElement.addEventListener('click',function(event){
    console.log(Object.keys(event));
},false);

//create simulated mouse event 'click'
var simulateDivClick = document.createEvent('MouseEvents');

/*setup simulated mouse 'click'
initMouseEvent(type,bubbles,cancelable,view,detail,screenx,screeny,clientx,clienty,ct
rlKey,altKey,shiftKey,metaKey,button,relatedTarget)*
simulateDivClick.initMouseEvent('click',true,true,document.defaultView,0,0,0,0,0,fals
e,false,false,0,null,null);

//invoke simulated clicked event
divElement.dispatchEvent(simulateDivClick);

</script>
</body>
</html>
```

Notes

Simulating/triggering mouse events as of this writing works in all modern browsers. Simulating other event types quickly becomes more complicated and leveraging simulate.js or jQuery (e.g. jQuery `trigger()` method) becomes neccsary.

11.14 Event delegation

Event delegation, stated simply, is the programmatic act of leveraging the event flow and a single event listener to deal with multiple event targets. A side effect of event delegation is that the event targets don't have to be in the DOM when the event is created in order for the targets to respond to the event. This is of course rather handy when dealing with XHR responses that update the DOM. By implementing event delegation new content that is added to the DOM post JavaScript load parsing can immediately start responding to events. Imagine you have a table with an unlimited number of rows and columns. Using event delegation we can add a single event listener to the `<table>` node which acts as a delegate for the node or object that is the initial target of the event. In the code example below, clicking any of the `<td>`'s (i.e. the target of the event) will delegate its event to the `click` listener on the `<table>`. Don't forget this is all made possible because of the event flow and in this specific case the bubbling phase.

live code: http://jsfiddle.net/domenlightenment/BRkVL

```
<!DOCTYPE html>
<html lang="en">
<body>

<p>Click a table cell</p>

<table border="1">
    <tbody>
        <tr><td>row 1 column 1</td><td>row 1 column 2</td></tr>
        <tr><td>row 2 column 1</td><td>row 2 column 2</td></tr>
        <tr><td>row 3 column 1</td><td>row 3 column 2</td></tr>
        <tr><td>row 4 column 1</td><td>row 4 column 2</td></tr>
        <tr><td>row 5 column 1</td><td>row 5 column 2</td></tr>
        <tr><td>row 6 column 1</td><td>row 6 column 2</td></tr>
    </tbody>
</table>

<script>

document.querySelector('table').addEventListener('click',function(event){
        if(event.target.tagName.toLowerCase() === 'td'){ //make sure we only run code
 if a td is the target
                console.log(event.target.textContent); //use event.target to gain acc
ess to target of the event which is the td
        }
},false);

</script>
</body>
</html>
```

If we were to update the table in the code example with new rows, the new rows would responded to the `click` event as soon as they were render to the screen because the click event is delegated to the `<table>` element node.

Notes

Event delegation is ideally leverage when you are dealing with a `click`, `mousedown`, `mouseup`, `keydown`, `keyup`, and `keypress` event type.