Module 1: Frontend Technologies

Module Overview

In this module, students will be able to learn about frontend technologies such as HTML5, CSS3, Bootstrap, JavaScript and TypeScript.

Module Objective

At the end of this module, students should be able to demonstrate appropriate knowledge, and show an understanding of the following:

- Develop Static web page.
- Know all about latest technologies.
- Creating responsive, mobile first web projects.

HTML Basics

HTML Basics

What is the HTML?

HTML – Hypertext Markup Language is a markup language that is used to define how text should be displayed in a browser or other software that is capable of interpreting HTML, such as word processors. The basic structure of an

HTML document includes tags, attributes and elements. Every web page is actually a HTML file. Each HTML file is just a plain-text file, but with a .html file extension instead of .txt and is made up of many HTML tags as well as the content for a web page. A web site will often contain many html files that link to each other.

HTML tags are the hidden keywords within a web page that define how your web browser must format and display the content. HTML tags are element names surrounded by angle brackets. Most tags come in pairs, called the opening tag and the closing tag. An opening tag consists of tag name contained within angled brackets. A closing tag is similar in appearance to the opening tag except the closing tag has forward slash (/) before the tag name within the angled brackets. Text that appears between the opening and closing tag is displayed according to the definition of the tag. For example, the following tag pairs causes text between the tags to appear in bold. The **<B>** is the HTML tag for bold.

```
<B> First Name:</B>
```

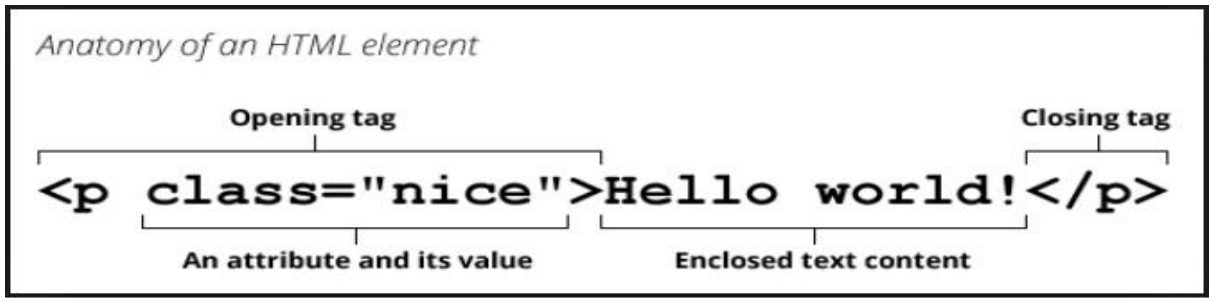
Elements can also have attributes that look like the following:

```
<FONT SIZE = “7”> First Name:</FONT>
```

The opening FONT tag contains the attribute SIZE, which is assigned the value 7. Attributes contain extra information about the element that you don't want to appear in the actual content. Here, SIZE is the attribute name and 7 is the attribute value.

An attribute should always have the following:

1. A space between it and the element name (or the previous attribute, if the element already has one or more attributes).
2. The attribute name followed by an equal sign.
3. The attribute value wrapped by opening and closing quotation marks, although quotation marks are optional if the value contains only letters, digits, a hyphen, or a period.



There are several common attributes that may appear in many elements:

- The id attribute provides a document-wide unique identifier for an element.
- The class attribute provides a way of classifying similar elements.
- The title attribute is used to attach sub-textual explanation to an element.

HTML Coding Basics & Selenium

Now how do you think Selenium interacts with the fields displayed on a webpage? ***How can a script click on a particular link?*** Or ***select a particular radio button?*** Or ***enter a text in the textbox?*** Or ***click on a particular button (say submit)?*** Yeah! The script first needs to identify the field in the webpage and then perform corresponding action, right?

The starting point to learn Selenium (or any other automation tool) is to identify & understand the way Selenium identifies these fields in a webpage. E.g. Textbox, Button, Image, Checkbox, Dropdown, Radio button, Hyperlink, Table, etc. – everything which is visible in a web application is known as "Element". For Selenium it's all about identifying these HTML "Elements" using its certain "Attributes" (known as "Locators" in Selenium) and then performing the required action to-be-performed on it. Selenium offers a wide choice of Locators (HTML Attributes), namely: ***ID, Name, Identifier, Link, XPath, CSS, Class, tag, DOM, Filters, etc.***

Types of Web Elements / HTML Elements:

We can divide the elements in three ways:

- Single Elements
  We can easily find out a locator for Single Element to work with selenium. We can locate by ID or by name or by linkText
- Group Elements
  We should try to prefer name as identifier along with a combination of value or index property for Group elements,
- Customized Elements
  Custom Elements allow web developers to define new types of HTML elements

Exercise

Trainer will initiate a discussion of common questions on HTML as given below:

1. **What does HTML stand for?**

   a. Home Tool Markup Language
   c. Hyper Text Markup Language
   b. Hyperlinks and Text Markup Language

2. **Choose the correct HTML element for the largest heading:**

   a. <h6>
   c. <h1>
   b. <heading>
   d. <head>

3. **Which character is used to indicate an end tag?**

   a. /
   c. *
   b. ^
   d. <

4. **What does HTML stand for?**

   a. <img src="pic_mountain.jpg"/ c. <img src="pic_mountain.jpg">

   b. <img "pic_mountain.jpg">

5. **What is the purpose of HTML?**

   a. To make a formatted document c. To make a webpage

   b. To edit photos d. To make a creative slide

Creating a Simple Page

Creating a Simple Page

We are done with the intro and we are ready to get our hands dirty. This is always the favourite part. So, here is the game-plan for moving forward:

1. We need to set up our development environment.
2. We need to get a quick tour of the basics of our development environment.
3. We build a basic Hello World web page (Woo-hoo!).
4. We deconstruct the web page to identify all the parts.

That will start us off well. From there, we will move into more HTML markup and then we will mix in our first CSS and see how that starts to change things.

Requirements

What do I need to start developing with HTML?

There are absolutely no requirements to start learning HTML, but you will need some tools to help you along the way. There are two tools that are essential to becoming an efficient and professional Web Developer.

Firstly, you will need a Text Editor.

Windows users, you can get an awesome text editor from notepad-plusplus.org. As you have probably guessed from the name of the URL, this text editor is Notepad ++ and includes some cool syntax highlighting!

Secondly, you will need a browser to render your code. You can use any browser that supports HTML5- Firefox, Safari, Google's Chrome and Opera.

Structure:

Let's start with a basic HTML5 structure:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>

    </body>
</html>
```

Exercise one:

1. Type out the above code into a new text file.
2. Save the file with an .html extension (i.e, structure.html).
3. Open the html file with your Browser.
4. What do you see?

No Output?

The Browser didn't seem to show us any content did it?

The browser would not show us any content (output), as we have not yet told the html document to output anything to the browser. All we have told the browser is that we have an HTML document.

Validation

Even though all we have created is a HTML structure and we are not seeing any results (yet), the HTML document we have should validate with W3C's online HTML (http://validator.w3.org/#_blank) validation tool.

W3C is the World Wide Web Consortium- they set the standards for HTML (and many other Web Based Languages) to provide a similar cross-browser experience; meaning that web browsers will be more inclined to output data in a similar fashion.

Exercise two:

1. Go to W3C's online HTML validation tool,
2. Select the third tab along- "Validate by Direct Input",
3. Copy and paste your HTML5 document code into the window and click 'Check'.
4. Notice how the document passed validation with three warnings.

Let's fix these warnings:
1) The warning- "Using Experimental Feature- HTML5 Conformance Checker" is basically telling us that all major browsers do not officially support HTML5 yet.
2) The next warning- 'No character encoding declared at document level'- this is because we haven't declared our character encoding within the HTML structure.
3) The final warning is telling us that no matter what our character encoding is set to within our HTML document that we are validating, it is going to assume, and treat is as UTF-8. The logical way to overcome this last warning is to use the file upload tool, rather than the Direct Input. Now, let's try and use the W3C validation tool to upload our HTML document, by selecting the second tab on the W3C's online HTML validation tool page and uploading our HTML document.

More Warnings:
If you notice we still have our 'Using Experimental Feature- HTML conformance checker' warning (which is perfectly fine, as we are using HTML5 and it is not yet fully supported by all browsers), the other 2 warnings are related to our character encoding and so is the Error we are now seeing.

Let's fix this, by ***declaring our Character Set***. We will be using the UTF-8 Character encoding and we can do this by adding some simple mark-up to our head section.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>

    </body>
</html>
```

We have just added our first HTML Meta tag! This Meta tag lets the browser know that we are using UTF-8 character encoding.
UTF-8 is the most commonly used character encoding, basically it provides a standard format (encoding) for text (code) that will assist against the problems of endianness, which could result in incorrect or invalid characters displaying (http:// en.wikipedia.org/wiki/Endianness).

Tags inside the head element of a HTML document are often used to tell the browser information about the HTML document that we don't need to output as part of our content, such as our HTML title and character encoding.

Comments in HTML

In every programming language comments are widely used to help remind other developers what is happening in the code, to make note of extra code that will be added to the web application at a later date and notes to others who may be working on the same project.
In HTML, comments are easily added to the document, by adding the opening and closing comment tags.

```
<!--
    This text will not be rendered by the browser
-->
```

Our browser will not render anything placed inside the comment tag.

Example of Comments:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
        <!--
            This is a comment
        -->
    </body>
</html>
```

HTML5 Structure

Let's go through the HTML document below and talk about the structure step-by-step, using comments.

```
<!DOCTYPE html>
<!-- declaring the document type -->
<html>
<!-- opening html tag -->
    <head>
    <!-- opening head tag -->
        <meta charset="UTF-8">
        <!-- declaring our character set -->
        <title>Our Title</title>
        <!-- opening and closing title tags -->
    </head>
    <!-- closing head tag -->
    <body>
    <!-- opening body tag -->
        <!--
            This is where we put our content
        -->
    </body>
    <!-- closing body tag -->
<!-- closing html tag -->
</html>
```

Exercise three (Part 1):

1. Type the above code into a new file.
2. Save the file with an .html extension (i.e, template.html).
3. Open the html file with your Browser.
4. What do you see?

Nothing has been output by the Browser, as we have used comments to explain the structure and have not yet added any “real” content.

Inspecting with Developer tools

Exercise three (Part 2):

1. Open up the Developer tools (ctrl + shift + I). You can do this by right clicking on the Browser window and selecting "Inspect".
2. Notice how we can see exactly what we have typed into our HTML document on the left-hand side of the window.

So how can we tell the browser to output some data?
It's actually quite simple. But before we add any content to the document, let's talk about the title tag. The title tag allows us to specify the name of the website- more specifically, the web page. It is good practice to be as relevant as you can when giving your web page a title.

As you can see in the previous examples, the title tag is inserted into the head section of the HTML document.

```
<title>Title of the Webpage</title>
```

If we load up this HTML document in our Browser now, we won't see any changes to the webpage. But, have a look at the top of your browser window or current tab- this is where the Title of your HTML document is shown.

Adding Content

We can add our content in-between our body tags like so:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Hello World</title>
    </head>
    <body>
        Hello World in HTML5
    </body>
</html>
```

Exercise four:

1. Type the above code into a new file with your text editor.
2. Save the file with an .html extension (i.e, hello_world.html)
3. Open the html file with your Browser and see what the browser is rendering.
4. Remove the “Hello World” text and replace it with a sentence about your favourite season and start to get comfortable with coding in HTML.
5. Make sure you view your html document in your browser and validate it.

Notes About HTML

- All versions of HTML require the basic HTML structure, but the version of HTML depicts a vast difference in the required elements and doc type declarations. HTML4.01, XHTML and HTML5.
- Notice how every tag is closed in order of which they were opened? This is a very important element to valid HTML.
- HTML5 is not currently supported by all major browsers, but provides plenty of extra features for us to work with and stay ahead of the curve. Although all major browsers do not support HTML5, Google's Chrome, Opera and FireFox are currently the most useful tools for Modern Web Development.

HTML Elements

HTML Elements

The Paragraph Tag

In HTML, the paragraph tag is part of the block level elements group. Block level elements will generally start on a new line and any Mark-up under or after the block level element will also start on a new line.

Here is an example of a paragraph tag in HTML, followed by some text after the ending paragraph tag. Even though all of the text is on one line, the paragraph tag (block level element) will place the text after the closing paragraph tag on a new line.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Block Level Elements</title>
    </head>
    <body>
        <p>I am a paragraph</p>I am directly after the paragraph
    </body>
</html>
```

Output:

I am a paragraph

I am directly after the paragraph

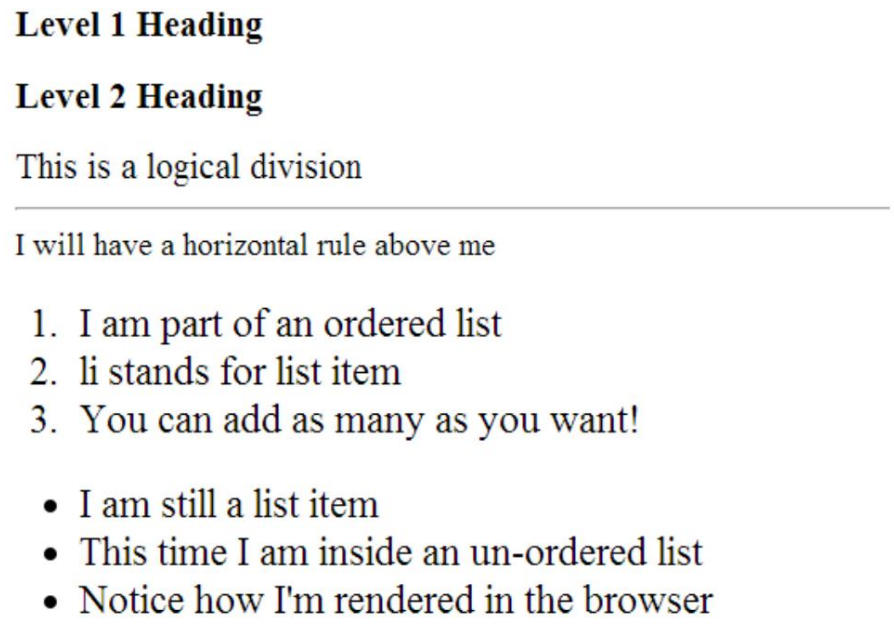
Other Block Level Elements

There are a variety of other block level elements available in HTML; including ***Headings***, ***logical*** (or document) ***divisions***, ***horizontal rules***, ***ordered lists*** and ***unordered lists***.

So, let's check out some of these block level elements in action.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Block Level Elements</title>
    </head>
    <body>
        <h1>Level 1 Heading</h1>
        <h2>Level 2 Heading</h2>
        <div>This is a logical division</div>
        <hr>
        I will have a horizontal rule above me
        <ol>
            <li>I am part of an ordered list</li>
            <li>li stands for list item</li>
            <li>You can add as many as you want!</li>
        </ol>
        <ul>
            <li>I am still a list item</li>
            <li>This time I am inside an un-ordered list</li>
            <li>Notice how I'm rendered in the browser</li>
        </ul>
    </body>
</html>
```

Output:



Exercise five:

1. Take note of the code above.
2. Type out the code into a new html document.
3. Change the content of the code to be about your favourite dessert.
4. Add an extra ordered list (containing 7 list items) and a logical division (containing a paragraph element) to the end of the page.
5. Save the html document as “block_level_elements.html”.
6. Open the document with your browser and make sure it appears as intended.
7. Upload the html document to the online validator and correct any warnings using the skills you have learned thus far.

Line Breaks vs. Paragraphs

The break element or tag in HTML (as you can guess) provides a line break (or new line). Some people may like to 'over-use' this element, but I suggest using the paragraph element when dealing with text (where possible), to provide formatting and appropriate spacing.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>The Break Tag</title>
    </head>
    <body>
        This text is on one line
        <br>
        This text is on another line
    </body>
</html>
```

We could do the same thing with the paragraph tag, but with a better format.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Block Level Elements</title>
    </head>
    <body>
        <p>This text is on one line</p>
        <p>This text is another line</p>
    </body>
</html>
```

Exercise six:

1. Type out the above code into a new html document.
2. Add some additional break tags and paragraph elements containing text.
3. Save the file with as “break.html”.
4. Open the html file with your Browser and take note of how each tag performs almost the same task.

Inline Elements

Now we have an understanding of what block level elements are, it’s time to move on to some inline elements.
Text Modifiers- Introducing the ***strong*** and ***em*** tags.
As you may have guessed, ***the strong tag*** is used to define important text and will render text as bold.
The em tag is a little harder to guess. The em tag renders text as Italic and is used to 'emphasize' text. The Strong and em tags are both part of the Text Modifiers group.

Example:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Text Modifiers</title>
    </head>
    <body>
        The <strong>strong tag</strong> will render bold text.
        <br>
        The <em>em tag</em> will render italic text.
    </body>
</html>
```

Text modifiers can be a simple way to make certain text stand out or add character to a document. Just *like* **this!**

Images

Images are an important part of any form of content, especially websites. As a web developer, you will find it very helpful and necessary to be able to place images onto a web page.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Images</title>
    </head>
    <body>
        <img src="" alt="">
    </body>
</html>
```

We would then put the URL for our image inside the src (source) attribute. The URL could be relative or absolute.

Here is an example of using an absolute URL with the src attribute of the img tag:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Images</title>
    </head>
    <body>
        <img src="http://glorified.me/image.png" alt="">
    </body>
</html>
```

When working in a Live or Development environment it is good Practice to use relative file Paths, rather than absolute or full file Paths.

• A relative file Path can be defined as being a localised link, using the current directory Structure as a means of navigation between files.
• An absolute file Path is a direct link or URL to a file.

If we had an image titled 'logo.png' in the same folder or directory as our Current html file, we could simply link to that file just by using the files name:

```
<img src="logo.png" alt="">
```

If our image or file were in a directory titled "images" inside our Current folder or directory we would then link to the Image using

```
<img src="images/logo.png" alt="">
```

Sometimes we need to navigate downwards (as opposed to upwards) in our directory structure. If our directory Structure looked something like:

/home/html/Public/Current/

And our Current html document is in our "current" folder; we could link to our Image (which Could be located at /home/html/Public/Images/) by using:

```
<img src="../images/logo.png" alt="">
```

../images/ basically tells the browser to navigate one directory down and then into our images directory.

Alternate Text

When an image is unable to be displayed by a browser, we need a fallback method. So, the alt (alternate text) can be used as our fallback method- meaning we will have some descriptive text to display if the image itself is unable to be displayed for any reason.

An example of an image not displaying could be a HTML email (Gmail will, by default hide any images and ask the user if they want to show images) or the results in a search engine. Search Engines cannot "read" images, so they can only render "alternate" text in Search Engine Result Pages (SERPs).

It is good to be descriptive and short with the alt attribute, like so:

```
<img src="../images/logo.png" alt="HTML5 Logo">
```

Displaying an Image

So, let's try out our image tag with a real image!

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Images</title>
    </head>
    <body>
        <img src="html5-logo.png" alt="HTML5 Logo">
    </body>
</html>
```

Output:

I feel as though it would be a better idea to have the image a little smaller, wouldn't you agree?

Specifying Size of the Image

We can specify both height and width attributes inside our image tag like so:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Images</title>
    </head>
    <body>
        <img src="html5-logo.png" width="100" height="100" alt="HTML5 Logo">
    </body>
</html>
```

Note: *The height and width values are in px (pixels).*
So, let's try out this image with the height and width attributes specified!

I am going to guess... It looks a little more appropriate now!

Exercise seven:

1. Create a new HTML5 document.
2. Using the skills you have learned so far, add an image (of your choice) with a width and height of your choosing to your HTML5 document.
3. Load your HTML5 document in your browser and make sure it renders as expected.
4. Head over to the W3C's HTML Validator and validate your HTML5 Document. Correct any errors you receive.

Anchor Tags/ Hyperlinks

Now, let's move on to the ever-so important anchor tags.

We use an anchor tag like so:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Anchor Tags</title>
    </head>
    <body>
        <a href="http://glorified.me">This is a Hyperlink</a>
    </body>
</html>
```

The above code will render as:

This is a Hyperlink

Let's try a simple link to Google:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Anchor Tags</title>
    </head>
    <body>
        <a href="http://google.com">Go to Google?</a>
    </body>
</html>
```

The above example will render:

Go to Google?

Type out the above code and Try it out, notice that the browser will now load Google's homepage when you click on the link.

We have covered how to link to a page, but what if we want our users to go to our linked page, but in a new window (so they don't leave our interesting website)?

We can do just this by using the target attribute of the anchor tag and passing in a value of '_blank'.

Opening in a New Window:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Anchor Tags</title>
    </head>
    <body>
        <a href="http://google.com" target="_blank">Go to Google?</a>
    </body>
</html>
```

The above example will render:

Go to Google?

Type out the above code and Try it out!
Notice that the browser will now load Google's homepage in a new window when you click on the link.

Hypertext Reference

The most important attribute for the anchor tag is the href attribute. The href (Hypertext Reference) attribute will tell the anchor tag where to link to, or where to send the user once clicked on.
This example is slightly different to the previous examples:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Anchor Tags</title>
    </head>
    <body>
        <a href="#">Where to?</a>
    </body>
</html>
```

Linking inside a HTML document

Basically, the '#' symbol can also act as a page anchor, when nothing is assigned to the '#' in the href attribute of an anchor tag- when the user clicks on it, as the link does not have a specific location- it will generally go nowhere. Now, we can assign id attributes to some of our elements, to use our anchor tags to link to specific parts in our HTML, rather than just having the '#' symbol, we would use:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Anchor Tags and Ids</title>
    </head>
    <body>
        <div id="top">Top of the Page!</div>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin ultricies ligula eget lorem
malesuada euismod. Phasellus ac interdum mauris. Vivamus hendrerit auctor arcu, vitae iaculis sem
fringilla eget. Pellentesque adipiscing erat at mollis pellentesque. Aenean fringilla lectus et massa
laoreet hendrerit. Proin vehicula ante non mi molestie semper. Vivamus rutrum elit at sem mattis,
quis mollis neque dapibus. Aliquam sit amet justo nunc. Nunc vel pharetra quam. Morbi accumsan velit
at aliquam lobortis. In luctus, ipsum et accumsan viverra, nisi ipsum tristique enim, ut pellentesque
dolor leo quis nisi. Pellentesque porttitor ultrices sollicitudin. Cras ut odio erat. Duis gravida
pretium tellus, lacinia rutrum sapien vestibulum at.

        Proin vestibulum ante mattis, rutrum quam vel, congue nisl. Ut ornare dignissim urna,
consequat sollicitudin urna. Proin condimentum quis risus nec ultricies. Nulla tincidunt massa
ligula, vel volutpat magna consequat eget. Praesent viverra metus a venenatis mollis. Donec laoreet
fermentum lacinia. Donec vitae sodales mauris.
        <a href="#top">Top of Page?</a>
    </body>
</html>
```

I have added a logical or document division at the start of the above example with an id of top.
Ids are a very useful feature of HTML, but for now we are just going to use it for an anchor link (something to link to). We could assign our div with any id value, as long as we reference it in our href attribute of our anchor tag.

Exercise eight:

1. Create a new HTML document and save it as anchors#.html.
2. Add a logical division with an id of ‘top’ and an anchor tag with a href value of ‘#top’.
3. Head on over to lipsum.com and generate some ‘dummy text’, copy it and paste it in between the logical division and the anchor tag you have placed in your HTML document. Save your document.

4. Open your anchors#.html document in your browser, scroll down to the bottom of the page and click on the link you have created.
3. Notice how the link takes you to the top of the page (The logical division with an id of 'top').

Note: *Make sure that there is enough text to actually cover the height of the page.*

Email Links

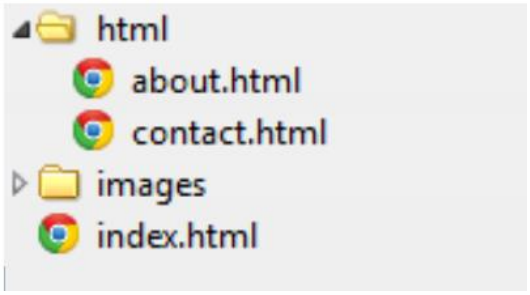
In HTML we can create a matilto: link, when the user clicks on this link their email client will open and the mailto: value (our email address) will be added to the TO: field.

```
<a href="mailto:admin@glorified.me?Subject=Email">Contact Us</a>
```

This makes it easy and enticing for a user to quickly send us some email, regarding our web page.
You may have noticed that I have added '?Subject=Email', this will add "Email" to the subject field within the email.
You can change the mailto and Subject values to suit your needs.

Directories

You will often be using a folder structure. The folder structure is a crucial part of good coding practice and helps to tidy up our files (an images folder and an html folder).
A basic html folder structure would look similar to this:



Note: *When working with folder in Web Development, we refer to folders as directories.*

As you can see, we have a few html files in different locations. We have our index.html file, which; when working in a server environment will automatically load once we navigate or load the specific folder that index.html resides in. As we are not working in a server environment this is not required knowledge now.

We have an about.html and a contact.html file in our “html” directory. The content of these two files are irrelevant at this point in time. The purpose of the following exercise is to make sure you have a grasp of 'navigating the directory structure'.

Example of linking to the about.html page from index.html:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Anchor Tags</title>
    </head>
    <body>
        <a href="html/about.html">About Us</a>
    </body>
</html>
```

Exercise nine:

1. Create three new html documents named; about.html, contact.html and index.html.
2. Create the same directory structure as shown above and place your new html files in the appropriate directories.
3. Give each html file an appropriate title tag and a level 1 heading (h1).
4. Add a paragraph of appropriate text to each of our html pages and a mailto: link to the contact page.
5. Now that we have some text and a heading for each html document, we need place an image (you can use any image you would like) on each page.
6. Using your skills, you can add an image under the paragraph tag to each html document and a link to each other html document in the directory structure under the image.
7. Test out your html documents by saving them with the same names as detailed above. Make sure the images in your html documents are appearing in the browser and when you click the links under the images, you are taken to the respective html document (i.e. a link to about.html should take you to the about.html page when you click on the link) in your browser.

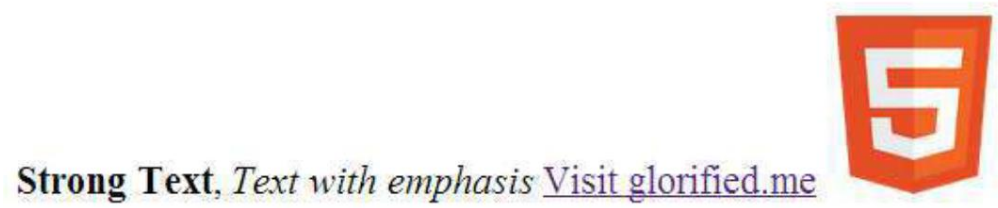
8. Now that you have completed your three html pages, go ahead and validate them with the online validator and fix any errors that appear.

Inline Elements in Action

Let's check out these inline elements in action.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Inline Elements</title>
    </head>
    <body>
        <strong>Strong text</strong>, <em>Text with emphasis</em>
        <a href="http://glorified.me">Visit glorified.me</a>
        <img src="html5-logo.png" width="100" height="100" alt="HTML5 Logo">
    </body>
</html>
```

Output:



Notice how the contents of the strong, em, anchor and image tags are being displayed on the same line, rather than being displayed on separate lines- like the block level elements.

Inline & Block Elements in Action

So, let's try some block level and inline elements together.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Block Level and Inline Elements</title>
    </head>
    <body>
        <h2>Level 2 Heading</h2><p>A Paragraph</p><em>and some text with emphasis</em>
        <a href="http://glorified.me">Visit glorified.me</a>
        <img src="html5-logo.png" width="100" height="100" alt="HTML5 Logo">
        <p>Another paragraph</p> and some <strong>strong</strong> text.
    </body>
</html>
```

Output:

Level 2 Heading

A Paragraph

and some text with emphasis Visit glorified.me

Another Paragraph

and some **strong** text.

Tables

Sometimes when we have some information to display on our web page, it makes sense to display that information or data in a table. Tables in HTML are relatively simple. We have the Opening Table Tag and the closing Table tag. Inside of our Table element, we have table rows. Inside the table rows we have tabular data or cells. But, for our table headers, we will be using the table header tags inside our table row.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Tables</title>
    </head>
    <body>
        <table>
            <caption>My HTML5 Table</caption>
            <!-- the caption tag is new in HTML5 -->
            <tr>
                <th>Color</th><th>Width</th><th>Height</th>
            </tr>
            <tr>
                <td>Black</td><td>55</td><td>99</td>
            </tr>
            <tr>
                <td>Green</td><td>85</td><td>45</td>
            </tr>
        </table>
    </body>
</html>
```

Forms

One of the many things you may have noticed on a web page is a contact form for example. We can create a form in HTML by using the form opening and closing tags. Inside of our Form element we have inputs and a submit button. Some of our inputs will be of type text and our submit button will be an input of type submit.
When we work with HTML forms in the real world, there are two attributes that we need to add to our opening form tag.

The scope of this module will not be covering server-side processing of HTML data, but it is helpful to know what these attributes do.

For the action attribute, you will enter in the destination of the Form data. The method will take either a POST or GET value. POST and GET are used with server-side processing.

For the action attribute, you would enter in the destination of the Form data. The method will take either a POST or GET value. You would generally be using POST to submit most forms of data. The main difference between POST and GET is that POST sends data to the server 'behind the scenes', whereas GET will form a query string. A query string

consists of name attribute values and the values input by the user. As you can imagine, if we were submitting sensitive data to the server we would definitely not use GET; in the case that we did, all information input by the user would be clearly visible in the URL address bar.
For the purpose of this module, we will be taking a look at forming a query string, using GET.

Type out the following code and select some values from the form and click the submit button. Take a look in your URL Address bar; this is called a query string- the part following the '?'.

```
<form action="#" method="GET">
    <label for="option1G1">Radio Button 1:</label>
    <input id="option1G1" type="radio" name="radio1" value="option1G1">
    <br>
    <label for="option2G1">Radio Button 2:</label>
    <input id="option2G1" type="radio" name="radio1" value="option2G1">
    <br>
    <label for="option1G2">Radio Button 3:</label>
    <input id="option1G2" type="radio" name="radio2" value="option1G2">
    <br>
    <label for="option2G2">Radio Button 4:</label>
    <input id="option2G2" type="radio" name="radio2" value="option2G2">
    <br>
    <label for="cb1">Checkbox 1:</label>
    <input id="cb1" type="checkbox" name="cb1" value="cb1">
    <br>
    <label for="cb2">Checkbox 2:</label>
    <input id="cb2" type="checkbox" name="cb2" value="cb2">
    <br>
    <input type="submit">
    <input type="reset">
</form>
```

Again, type out the following code, load it up in your browser, enter some text into the inputs in the form and click the submit button. ***Take a look at your URL Address bar.***

```
<form action="#" method="GET">
    <label for="first">First Name: </label>
    <input id="first" type="text" name="first">
    <br>
    <label for="last">Last Name: </label>
    <input id="last" type="text" name="last">
    <br>
    <label for="password">Password: </label>
    <input id="password" type="password" name="password">
    <br>
    <input type="submit">
    <input type="reset">
</form>
```

The label tag allows us to add descriptive text regarding the input expected from the user and when the user click the label text, focus will be drawn to the input. To use labels properly, you must give the label a 'for' attribute and a value of the 'for' attribute that corresponds to the input's id.

When you use an input of type password, you may think that because the input entered renders as dots that it must be secured. This is a common misconception.
The dots that render for password inputs are only a masking mechanism. Meaning that the input is still just the plain text that the user enters, but the password field will mask this to prevent prying eyes when a user is entering in a password. By using GET, you have just seen (in the query string) that the input entered into a password field remains plain text.

Exercise ten:

1. Create a new .html file titled tables-forms.html.
2. Add a table (4x7) and type in 4 of your favourite bands and seven of their best songs, to fill out the table. By doing this simple exercise, you will learn how to create a HTML table of any size.
3. Under the new table that you have created; add a form.
4. This form will have 4 radio inputs, 3 Check boxes, 1 text and 1 password input field.
5. Appropriately name each input and test that you have done so correctly by entering in text/ making selections and submitting the form (clicking 'submit'). Pay close attention to the query string that has formed in your URL address bar.
6. Create some new inputs of type: colour, number, URL, date and email and test them out in your browser!

HTML5 Specific Elements

So far, we have only really learned about general HTML tags and elements, now it's time to get into some HTML5 specific elements. As you may notice when you start to work with HTML and build some of your own web pages, it would be really helpful if there was a logical way to define the header and footer of our web page With HTML5, we can just that with the header and footer tags:

Header & Footer

```
<header>This is the Header Element</header>
```

```
<footer>This is the Footer Element</footer>
```

The header element defines a header for our html document. We could also have multiple headers in our html document, to define headers for different parts of our html.

The footer element defines a footer for our html document; just as we can with the header element we can also have multiple footer elements within our html document defining footers for different parts of our html.
Along with these new header and footer elements, there are also a few more important elements that have been introduced with HTML5.

Navigation

We can now define a navigational element with the nav tag:

```
<nav>This is the Nav Element</nav>
```

We would use this element to hold our main navigational content.

Section

We can now define a ‘section’ of our HTML document. Do keep in mind that the contents of a section should be related.

```
<section>This is the Section Element</section>
```

Article

We can now define an 'article' within our HTML document. Within a section, we could have several articles:

```
<section>
    <article>
        This is the Article Element
    </article>
    <article>
        This is another Article Element
    </article>
</section>
```

Aside

We can now define an 'aside'; a part of our HTML content that is 'aside' from our main content, but is still related:

```
<article>
    This is another Article Element
    <aside>
        This is the Aside Element
    </aside>
</article>
```

The Meter Element

The ***<meter>*** tag defines a scalar measurement within a known range, or a fractional value. This is also known as a gauge.
One of the really cool things with HTML5 is the ability to add meters. We define the meter element with the opening and closing meter tags. The HTML5 meter tag will take a few attributes, min, max and value.
The Min and Max values will define a range in which our value will be compared. For example, with a min of 0 and a max of 100, a value of 50 will show a 'gauge' that is 50% complete:

```
<meter min="0" max="100" value="50">This is the Meter Element</meter>
```

The text that I have entered in between the opening and closing meter tag is fallback text. This fallback text will be rendered in older browsers that do not yet support the meter tag.

The new HTML5 Elements in Action

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>HTML5</title>
    </head>
    <body>
        <header>This is the header element</header>
        <nav>This is the nav element</nav>
        <section>
            <article>
                This is the article element
                <aside>
                    This is the aside element
                </aside>
            </article>
            <article>
                This is another article
            </article>
            <article>
                And, yet another article
            </article>
        </section>
        <footer>This is the footer element</footer>
    </body>
</html>
```

Exercise eleven:
1. Using only the new HTML5 tags, create a Valid HTML5 document with some content about your favourite holiday.
2. Validate this HTML5 specific document with the online validator and fix any errors.

Video

One of the greatest features of HTML5 is the ability to implement a fully featured Video with Controls.

We start out with the opening Video tag, passing in a height & width and the controls attribute, so the video player will have controls. Next we need to add the source tag and actually add the Video URL to the src attribute of the source tag. Again this can be an absolute or relative URL.

Along with our actual video src, we need to tell the browser the mime-type. A mimetype basically lets the browser know what kind of format or media type to expect.

To provide a similar cross-browser experience you will need to supply multiple formats, contained within separate source tags.
Here we have the video Element set to 640x480px with controls displayed, some fallback text (to display some text in a browser that doesn't support the video element) and two sources- each have a different format and mime-type.

```
<video width="640" height="480" controls>
    <source src="video.mp4" type="video/mp4">
    <source src="video.webm" type="video/webm">
    Fallback Text
</video>
```

Audio

As you have just learned about HTML5 Video, HTML5 Audio isn't going to be all that difficult to grasp. HTML5 audio isn't all that different from HTML5 Video (apart from the fact that it's audio and not video).

With the audio element, we do not need to set the size, but it is ALWAYS good practise to use the controls attribute so your users can actually operate the Audio Player.

Just as we can have multiple sources in our video element, we do the same thing with our audio element, passing in audio files and appropriate mime-types.

```
<audio controls>
    <source src="audio.mp3" type="audio/mpeg">
    <source src="audio.ogg" type="audio/ogg">
    Fallback Text
</audio>
```

The Basics of CSS

Definition of CSS

CSS stands for Cascading Style Sheets and provides HTML with layout and design.

Along with making things pretty and aesthetically pleasing, CSS also provides a general structure to HTML.

Some of the most important CSS properties (in my opinion) are (in no order):

- Color - specifying text color.
- Font-family - specifying font type.
- Font-size - specifying font size.
- Text-decoration - specifying text decorations, such as underline.
- Font-style - specifying font styling, such as italics.
- Font-weight - specifying font weight, such as bold.
- Width - specifying the width of an element.
- Height - specifying the height of an element.
- Background - specifying the background.
- Border - specifying a border.
- Text-shadow - specifying a shadow for our text.
- Float - specifying the float of an element, such as left or right.
- Position - specifying the position of an element, such as absolute or relative.
- Z-index - specifying the z-index of an element, such as 999; which would put that styled element 'on-top' of all other elements that either have a negative z-index specified or no z-index specified.
- Padding - specifying padding inside an element, such as padding around text.
- Margin - specifying the margin between elements.

CSS can be implemented in three different ways to our HTML:

1. Inline
2. Internal
3. External

Using Inline CSS

So, let's use some inline CSS to change a few things in our HTML structure.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Inline CSS</title>
    </head>
    <body>
        <header>
            <h1 style="color:red">My Heading</h1>
        </header>
        <section>
            <article style="color:blue">
                My Article of Content
            </article>
        </section>
        <footer>
            <strong style="color:green">My Bold Text</strong>
            <br>
            <em style="font-weight:100">My Company</em>
        </footer>
    </body>
</html>
```

Output:

Color

As you have probably noticed, we have used the English word for the color that we want to use. But there are two other ways we can define colors in CSS; the rgb color values and something called Hexadecimal.
All three types of defining colors in CSS are acceptable; you can read more about colors in CSS here:
http://www.w3schools.com/cssref/css_colors.asp

We will be using a mixture of the three different ways to define colors in CSS.

Using Internal CSS

As you can see, our inline CSS is very effective, but perhaps not very efficient. Inline CSS is good for adding slight changes or specifying colors for different text elements, but it starts to get a little 'wordy' and messy.
When we add CSS to HTML either; externally or in the head section, we can use selectors.

Selectors

Selectors allow us to 'select' or 'point' to a specific element of our HTML. CSS can use HTML elements as selectors, such as the paragraph, anchor, em and strong tags. If we referred to these elements as selectors in our CSS we would be styling every paragraph, anchor, em and strong element in our HTML.

Let's try the same thing, but this time adding our CSS to the head section of our HTML document and using selectors.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Internal CSS</title>
        <style>
            h1{
                color: #ff0000;
            }
            strong{
                color: #00ff00;
            }
            em.bottom{
                font-weight: 100;
                font-size: 1.1em;
            }
        </style>
    </head>
    <body>
        <header>
            <h1>My Heading</h1>
        </header>
        <footer>
            <em>My Italic Text</em>
            <strong>My Bold Text</strong>
            <br>
            <em class="bottom">My Company</em>
        </footer>
    </body>
</html>
```

I have added an additional em tag, to demonstrate using classes as selectors in CSS.

Using ids in CSS

As you may have guessed, we are using a class to identify our bottom em tag. The dot notation before the class name allows us to select or target an element in our HTML by its class name.
We can use ids like so:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Internal CSS</title>
        <style>
            h1{
                color: #ff0000;
            }
            strong{
                color: #00ff00;
            }
            em#bottom{
                font-weight: 100;
                font-size: 1.1em;
            }
        </style>
    </head>
    <body>
        <header>
            <h1>My Heading</h1>
        </header>
        <footer>
            <em>My Italic Text</em>
            <strong>My Bold Text</strong>
            <br>
            <em id="bottom">My Company</em>
        </footer>
    </body>
</html>
```

All we have to do is change 'class' to 'id' for the element we are referring to, and change the '.' in front of our CSS selector (in the head element) to the '#' symbol.

The # symbol (when not used as href attribute) is generally used to signify an id within the HTML. The major different between using classes and id's is; classes can be re-used time and time again in the same HTML document, whereas id's can only be used once in a single HTML document. You can think of a class as a group or multiple items, and an id as a single identification.

The output of the above code is the same (we have also set a font-size for the #bottom em tag) as the output for the previous code example, we are getting the same results as we are basically telling the browser the same thing, just in a different way.

There are several ways to make selectors 'unique' or point to only 'some' parts of the HTML.

A class is an effective way of referencing a specific part of our HTML; we can basically pinpoint the section of our code that contains the content we wish to style.

Note: When using em in CSS it's slightly different to the em tag in HTML. In HTML the em tag renders italic text. In CSS the em value can be used as a unit of measurement. A font size with a value greater than 1em will generate text larger than the default for that web page or User Agent, but does not render text as italic.

Ids must be unique, we can only use the same id only once in our HTML page. With classes, we can 'reuse' the class several times in our HTML page.

Creating External CSS

To add an external CSS to our HTML, we need to tell the HTML all about it- what relation it has to our HTML, the type of file it is and its location and name.

Remember the Meta tags from before?

Well, this is implemented in the same way (completely different concepts), by adding a line of code into our head section of our HTML document. We use a rel value to tell HTML what the CSS file's relation is to the HTML, a type value to tell the HTML the type of file it is and a href value telling the HTML where the file is located and its name.

Note: CSS files have a file extension of .CSS

We can add an external style sheet to our HTML by using link tag. So, let's create a small CSS file, to use externally.

```
header{
    color: rgb(255,0,0);
    font-size: 2em;
    text-decoration: underline;
}
section article{
    font-weight: 200;
    font-size: 1.1em;
}
footer{
    font-size: 0.8em;
}
```

Go ahead and save the above styling into a new CSS file, titled style.css.

Linking to External CSS

If our style sheet (CSS) were located in the same directory (or folder) as our HTML file, we would add the tag to the head section of the HTML document, like so:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>External CSS</title>
        <link rel="stylesheet" type="text/css" href="style.css">
    </head>
    <body>
        <header>
            <h1>My Heading</h1>
        </header>
        <section>
            <article>
                My Article of Content
            </article>
        </section>
        <footer>
            <em>My Footer</em>
        </footer>
    </body>
</html>
```

Just as our CSS example before, no matter of its location (inline, internal and external), the CSS will tell the browser to render the styles for our HTML in the same way.

Inefficient Selectors

Let's have a look at some inefficient CSS selectors with some external CSS:

```
header{
    color: red;
    font-size: 2em;
    text-decoration: underline;
}

section article{
    font-weight:200;
    font-size: 1.1em;
    color: red;
}

footer{
    font-size: 0.8em;
    font-weight: 200;
    color: red;
}
```

The advantages of using external CSS include the ability to completely separate the HTML from the CSS, to reduce individual file size and length, make things more readable and use effective selectors; meaning selectors that target multiple elements where necessary. Rather than having a large portion of CSS repeating and applying the same styling to different elements, we could 'join' the selectors together to create a smaller file size or to simply be more efficient with our use of CSS.

We can target or select multiple elements by separating the selectors with a comma in the CSS.
Let's fix this up!

Efficient Selectors

```
header{
    font-size: 2em;
    text-decoration: underline;
}

section article{
    font-size: 1.1em;
}

footer{
    font-size: 0.8em;
}

header, section article, footer{
     color: red;
}

section article, footer{
    font-weight: 200;
}
```

Exercise twelve:

1. Create your HTML5 document structure and create a file titled style.css
2. Add the new HTML5 elements and style the HTML accordingly:
- Articles within a section will have a font-size of 2 times the default font-size.
- The Footer and Header will have a text-decoration of underline and a color of red.
- The aside will have a font-weight of bolder and a color of blue.
3. Save your CSS and HTML files and load them into your browser, checking that they render as expected.

HTML Element State

With our new CSS abilities, we are able to style a HTML element, based on its 'state'. HTML Element state refers to the 'state' that the elements are in; some of these include: Hover and Active.

Hover:

You may have noticed that when you hover over a link on a web page, that the link will change color (among other aspects). We can do this with almost any HTML elements.

```
article:hover, article:hover a{
    background-color: #ff0000;
    color: rgb(255,255,255);
}
```

In the above example we have styled all 'hovered' over articles and the anchor tags, when the article is being 'hovered' over with a background of red and a text color of white.

Active:

Along with defining hover style, we can define active style. Active is defined by an element that is 'actively' being clicked on, i.e., if you are clicking a button, that button's state is now active.

```
article:active{
    background-color: #111111;
}
```

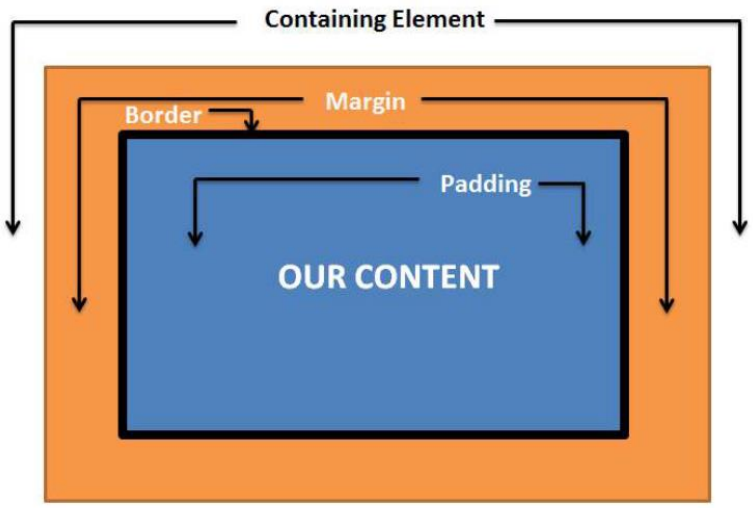
In the above example, an article that is 'active' will have a background color of almost black.

Exercise thirteen:

1. Create a new HTML document with an external Style sheet.
2. Add some styles that will target specific elements, based on their 'state'.
3. Make sure to make use of the rgb and hexadecimal color values.
4. Test your work in your browser and make sure it renders as expected.

The CSS Box Model

One of the fundamental understandings of CSS is the Box Model. The Box model helps us to understand the layout and design of HTML and CSS.
The CSS Box model is made up of content, Padding, Borders and Margins.



So, what are Padding, Margins and Borders?
As you can see, padding is the space that surrounds our content; borders are what surround the padding and margins are what surround the borders.

By definition:

- The padding is the area that separates the content from the border.
- The border is the area that separates the padding from the margin.
- The margin is the area that separates our box from surrounding elements.

How do we define these?

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>The Box Model</title>
        <style>
            section article{
                height: 50px;
                width: 500px;
                border: 1px solid #000000;
                padding: 50px 30px;
                margin: 0 auto;
            }
        </style>
    </head>
    <body>
        <section>
            <article>
                The CSS Box Model
            </article>
        </section>
    </body>
</html>
```

In the above example, we have set the padding for the top and bottom of the element to 50px and the left and right to 30px. The margin has been set to 0px for the top and bottom and the left and right margin is set to auto. When we set a value of auto in our margins, it will basically 'centre' the element within the containing or parent element.

As you may have noticed, we have also set our border. Our border is 1px wide for each side, is solid and has a color of black.

The above code renders the following output:

Note that the rendered output will be 152px in height (top and bottom padding, plus the width of our borders and the specified height of our element) and 562px in width (left and right padding, plus the width of our borders and the specified width of our element).

Exercise fourteen:

1. Create a new HTML file and an external Style sheet.
2. Using your knowledge of the CSS Box Model, create a 'box' that has a height of 60px, a width of 260px, a solid border of 3px (you can pick any color you wish), top and bottom padding of 55px, left and right padding of 25px and a top and bottom margin of 0px and a left and right margin set to auto.
3. Save, test your work in your browser and calculate the exact size of your box.

Fonts

When using CSS we can change the font-family of our text. We can specify multiple font-families for any given element. If the user has the first specified font on their system; that is the font that will be used. If the user does not have our first specified font on their system, the browser will attempt to render the next font and so on until one of the fonts are located on the users system. These font-families are separated with a comma and the proceeding fonts are referred to as fallback fonts.

```
header{
    font-family: "Helvetica Neue", Helvetica, sans-serif;
}
article, footer{
    font-family: Arial, sans-serif;
}
```

In the above example we are saying that our header should have a font-family of Helvetica Neue, if that is not located on the user's system, we will try Helvetica. If Helvetica is not located on the user's system, we will 'fall-back' to a generic sans-serif font.
You can find out more about sans-serif fonts here: http://en.wikipedia.org/wiki/Sansserif.

Main CSS3.0 Specific Properties

Opacity

In CSS3.0 we can easily specify opacity for an element:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Opacity</title>
        <style>
            section article{
                height: 100px;
                width: 200px;
                opactity: 0.5;
            }
        </style>
    </head>
    <body>
        <section>
            <article>
                We are setting the opactity for the article element.
            </article>
        </section>
    </body>
</html>
```

In the above code we are saying that every article within a section should have opacity of 50%. This will make all of our articles within a section appear transparent. When we set the opacity of an element, we are also setting the opacity of the contents as well and this can have undesired effects.

Alpha Color Space

Instead of using the opacity property in CSS, we can set an Alpha Color Space. We can do this by specifying the color or background-color of an element using the rgb color values.
Now, to specify the Alpha Color Space, we simply add an extra value to our rgb color values; what I mean by that is we change rgb to rgba and have four values for the colors, instead of the usual three.
The Alpha Color Space value will take a value of between 0 and 1. The Alpha Color Space will specify the opacity of that element.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Alpha Color Space</title>
        <style>
            section article{
                height: 100px;
                width: 200px;
                background-color: rgb(255,0,0);
                box-shadow: 2px 2px 3px #000000;
                border-radius: 8px;
                padding: 8px;
            }

            section article:hover{
                background-color: rgba(255,0,0,0.5);
            }
        </style>
    </head>
    <body>
        <section>
            <article>
                This is our article of content.
            </article>
        </section>
    </body>
</html>
```

In the above example we have set all of our articles within a section to have a background-color of red. When these articles are hovered over, we are setting using the same color, but with an Alpha Color Space value of 0.5. In other words, when we hover over our article elements we will have a background-color that will be slightly transparent.

In the above example you may have noticed the border-radius and box-shadow properties.

Box Shadow and Border Radius

We can specify a box-shadow for most of our HTML elements and this will literally give our 'box' (or element) a box-shadow; giving the element a 3D like appearance. The box-shadow property can take 4 arguments: the h-shadow value, the v-shadow value, the blur-distance and the color of the shadow.

The border-radius property will set a 'border radius' for each of our element's corners; resulting in rounded corners. This will be the resulting output when we hover over our element.

As you can see we have rounded corners, a shadow and we have an almost pink background-color. The background-color is set to red, but when we hover over it, we are setting the background-color to be 50% transparent.

Exercise fifteen:

1. Create a new HTML file and an external Style sheet.
2. Using all of the techniques, concepts and code that you have learned; create your first Website! It doesn't have to be glamorous! The point of this final exercise is to get you started with HTML5 and CSS3.0- in the real world!

JavaScript — Dynamic client-side scripting

JavaScript is a programming language that allows you to implement complex things on web pages. Every time a web page does more than just sit there and display static information for you to look at—displaying timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, or more—you can bet that JavaScript is probably involved.

JavaScript is the most popular and widely used client-side scripting language. Client-side scripting refers to scripts that run within your web browser. JavaScript is designed to add interactivity and dynamic effects to the web pages by manipulating the content returned from a web server.

JavaScript was originally developed as LiveScript by Netscape in the mid-1990s. It was later renamed to JavaScript in 1995 and became an ECMA standard in 1997. Now JavaScript is the standard client-side scripting language for web-based applications, and it is supported by virtually all web browsers available today, such as Google Chrome, Mozilla Firefox, Apple Safari, etc.

JavaScript is an object-oriented language, and it also has some similarities in syntax to Java programming language. But JavaScript is not related to Java in any way.

JavaScript is officially maintained by ECMA (European Computer Manufacturers Association) as ECMAScript. ECMAScript 6 (or ES6) is the latest major version of the ECMAScript standard.

What You Can Do with JavaScript

There are lot more things you can do with JavaScript.

- You can modify the content of a web page by adding or removing elements.
- You can change the style and position of the elements on a web page.
- You can monitor events like mouse click, hover, etc. and react to it.
- You can perform and control transitions and animations.
- You can create alert pop-ups to display info or warning messages to the user.
- You can perform operations based on user inputs and display the results.
- You can validate user inputs before submitting it to the server.

The list does not end here, there are many other interesting things that you can do with JavaScript.

Adding JavaScript to Your Web Pages

There are typically three ways to add JavaScript to a web page:

- Embedding the JavaScript code between a pair of <script> and </script> tag.
- Creating an external JavaScript file with the .js extension and then load it within the page through the src attribute of the <script> tag.
- Placing the JavaScript code directly inside an HTML tag using the special tag attributes such as onclick, on mouseover, on keypress, onload, etc.

The following sections will describe each of these procedures in detail:

Embedding the JavaScript Code

You can embed the JavaScript code directly within your web pages by placing it between the <script> and </script> tags. The <script> tag indicates the browser that the contained statements are to be interpreted as executable script and not HTML. Here's an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Embedding JavaScript</title>
</head>
<body>
    <script>
    var greet = "Hello World!";
    document.write(greet); // Prints: Hello World!
    </script>
</body>
</html>
```

The JavaScript code in the above example will simply prints a text message on the web page. You will learn what each of these JavaScript statements means in upcoming sections of this module.

Note: *The type attribute for <script> tag (i.e. <script type="text/javascript">) is no longer required since HTML5. JavaScript is the default scripting language for HTML5.*

Calling an External JavaScript File

You can also place your JavaScript code into a separate file with a ***.js*** extension, and then call that file in your document through the src attribute of the ***<script>*** tag, like this:

```
<script src="js/hello.js"></script>
```

This is useful if you want the same scripts available to multiple documents. It saves you from repeating the same task over and over again and makes your website much easier to maintain.

Well, let's create a JavaScript file named "hello.js" and place the following code in it:

```
// A function to display a message
function sayHello() {
    alert("Hello World!");
}

// Call function on click of the button
document.getElementById("myBtn").onclick = sayHello;
```

Now, you can call this external JavaScript file within a web page using the ***<script>*** tag, like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Including External JavaScript File</title>
</head>
<body>
    <button type="button" id="myBtn">Click Me</button>
    <script src="js/hello.js"></script>
</body>
</html>
```

Note: *Usually when an external JavaScript file is downloaded for first time, it is stored in the browser's cache (just like images and style sheets), so it won't need to be downloaded multiple times from the web server that makes the web pages load more quickly.*

Placing the JavaScript Code Inline

You can also place JavaScript code inline by inserting it directly inside the HTML tag using the special tag attributes such as ***onclick***, ***on mouseover***, ***on keypress***, ***onload***, etc.

However, you should avoid placing large amount of JavaScript code inline as it clutters up your HTML with JavaScript and makes your JavaScript code difficult to maintain. Here's an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Inlining JavaScript</title>
</head>
<body>
    <button onclick="alert('Hello World!')">Click Me</button>
</body>
</html>
```

The above example will show you an alert message on click of the button element.

Tip: *You should always keep the content and structure of your web page (i.e. HTML) separate out from presentation (CSS), and behavior (JavaScript).*

Positioning of Script inside HTML Document

The <script> element can be placed in the <head>, or <body> section of an HTML document. But ideally, scripts should be placed at the end of the body section, just before the closing </body> tag, it will make your web pages load faster, since it prevents obstruction of initial page rendering.

Each <script> tag blocks the page rendering process until it has fully downloaded and executed the JavaScript code, so placing them in the head section (i.e. <head> element) of the document without any valid reason will significantly impact your website performance.

Tip: *You can place any number of <script> element in a single document. However, they are processed in the order in which they appear in the document, from top to bottom.*

Difference Between Client-side and Server-side Scripting

Client-side scripting languages such as JavaScript, VBScript, etc. are interpreted and executed by the web browser, while server-side scripting languages such as PHP, ASP, Java, Python, Ruby, etc. runs on the web server and the output sent back to the web browser in HTML format.

Client-side scripting has many advantages over traditional server-side scripting approach. For example, you can use JavaScript to check if the user has entered invalid data in form fields and show notifications for input errors accordingly in real-time before submitting the form to the web-server for final data validation and further processing in order to prevent unnecessary network bandwidth usages and the exploitation of server system resources.

Also, response from a server-side script is slower as compared to a client-side script, because server-side scripts are processed on the remote computer not on the user's local computer.

JavaScript is case sensitive.

It should also be noted, before we begin, that JavaScript is extremely case sensitive so if you're trying to code along with any examples make sure lowercase is lowercase and uppercase is uppercase. For the most part JavaScript is also a camel-cased language. That is, if you're trying to express more than one word you will eliminate the spaces, leave the first letter uncapitalized and capitalize the first letter of each word. Thus "get element by id" becomes "getElementByID". By contrast, HTML itself is NOT case sensitive.

JavaScript Comments

A comment is simply a line of text that is completely ignored by the JavaScript interpreter. Comments are usually added with the purpose of providing extra information pertaining to source code. It will not only help you understand your code when you look after a period of time but also others who are working with you on the same project.

JavaScript support single-line as well as multi-line comments. Single-line comments begin with a double forward slash (//), followed by the comment text. Here's an example:

```
// This is my first JavaScript program
document.write("Hello World!");
```

Whereas a multi-line comment begins with a slash and an asterisk (/*) and ends with an asterisk and slash (*/). Here's an example of a multi-line comment.

```
/* This is my first program
in JavaScript */
document.write("Hello World!");
```

JavaScript Variables

What is Variable?

Variables are fundamental to all programming languages. Variables are used to store data, like string of text, numbers, etc. The data or value stored in the variables can be set, updated, and retrieved whenever needed. In general, variables are symbolic names for values.

You can create a variable with the var keyword, whereas the assignment operator (=) is used to assign value to a variable, like this: ***var varName = value;***

```
var name = "Peter Parker";
var age = 21;
var isMarried = false;
```

Tip: *Always give meaningful names to your variables. Additionally, for naming the variables that contain multiple words, camelCase is commonly used. In this convention all words after the first should have uppercase first letters, e.g. myLongVariableName.*

The let and const Keywords

ES6 introduces two new keywords let and const for declaring variables.

The const keyword works exactly the same as let, except that variables declared using const keyword cannot be reassigned later in the code. Here's an example:

```
// Declaring variables
let name = "Harry Potter";
let age = 11;
let isStudent = true;

// Declaring constant
const PI = 3.14;
console.log(PI); // 3.14

// Trying to reassign
PI = 10; // error
```

Unlike var, which declare function-scoped variables, both let and const keywords declare variables, scoped at block-level ({}). Block scoping means that a new scope is created between a pair of curly brackets {}. We'll discuss this in detail later, in JavaScript ES6 features chapter.

Note: *The let and const keywords are not supported in older browsers like IE10. IE11 support them partially. See the JS ES6 features chapter to know how to start using ES6 today.*

Naming Conventions for JavaScript Variables

These are the following rules for naming a JavaScript variable:

- A variable name must start with a letter, underscore (_), or dollar sign ($).
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters (A-z, 0-9) and underscores.
- A variable name cannot contain spaces.
- A variable name cannot be a JavaScript keyword or a JavaScript reserved word.

Note: *Variable names in JavaScript are case sensitive, it means $myvar and $myVar are two different variables. So be careful while defining variable names.*

JavaScript Generating Output

Generating Output in JavaScript

There are certain situations in which you may need to generate output from your JavaScript code. For example, you might want to see the value of variable, or write a message to browser console to help you debug an issue in your running JavaScript code, and so on.

In JavaScript there are several different ways of generating output including writing output to the browser window or browser console, displaying output in dialog boxes, writing output into an HTML element, etc. We'll take a closer look at each of these in the following sections.

Writing Output to Browser Console

You can easily outputs a message or writes data to the browser console using the console.log() method. This is a simple, but very powerful method for generating detailed output. Here's an example:

```
// Printing a simple text message
console.log("Hello World!"); // Prints: Hello World!

// Printing a variable value
var x = 10;
var y = 20;
var sum = x + y;
console.log(sum); // Prints: 30
```

Tip: *To access your web browser's console, first press F12 key on the keyboard to open the developer tools then click on the console tab.*

Displaying Output in Alert Dialog Boxes

You can also use alert dialog boxes to display the message or output data to the user. An alert dialog box is created using the alert() method. Here's is an example:

```
// Displaying a simple text message
alert("Hello World!"); // Outputs: Hello World!

// Displaying a variable value
var x = 10;
var y = 20;
var sum = x + y;
alert(sum); // Outputs: 30
```

Writing Output to the Browser Window

You can use the document.write() method to write the content to the current document only while that document is being parsed. Here's an example:

```
// Printing a simple text message
document.write("Hello World!"); // Prints: Hello World!

// Printing a variable value
var x = 10;
var y = 20;
var sum = x + y;
document.write(sum); // Prints: 30
```

If you use the ***document.write()*** method after the page has been loaded, it will overwrite all the existing content in that document. Check out the following example:

```
<h1>This is a heading</h1>
<p>This is a paragraph of text.</p>

<button type="button" onclick="document.write('Hello World!')">Click
Me</button>
```

Inserting Output Inside an HTML Element

You can also write or insert output inside an HTML element using the element's innerHTML property. However, before writing the output first we need to select the element using a method such as getElementById(), as demonstrated in the following example:

```
<p id="greet"></p>
<p id="result"></p>

<script>
// Writing text string inside an element
document.getElementById("greet").innerHTML = "Hello World!";

// Writing a variable value inside an element
var x = 10;
var y = 20;
var sum = x + y;
document.getElementById("result").innerHTML = sum;
</script>
```

Predefined Core Objects and Functions

Several objects are predefined in core JavaScript and can be used in either client-side or server-side scripts.

Array Object

JavaScript does not have an explicit array data type. However, you can use the predefined Array object and its methods to work with arrays in your applications. The Array object has methods for manipulating arrays in various ways, such as joining, reversing, and sorting them. It has a property for determining the array length and other properties for use with regular expressions.

An array is an ordered set of values that you refer to with a name and an index. For example, you could have an array called emp that contains employees' names indexed by their employee number. So emp[1] would be employee number one, emp[2] employee number two, and so on.

Creating an Array

The simplest way to create an array in JavaScript is enclosing a comma-separated list of values in square brackets ([]), as shown in the following syntax:

```
var myArray = [element0, element1, ..., elementN];
```

Array can also be created using the Array() constructor as shown in the following syntax. However, for the sake of simplicity previous syntax is recommended.

```
var myArray = new Array(element0, element1, ..., elementN);
```

Here are some examples of arrays created using array literal syntax:

```
var colors = ["Red", "Green", "Blue"];
var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];
var cities = ["London", "Paris", "New York"];
var person = ["John", "Wick", 32];
```

Note: *An array is an ordered collection of values. Each value in an array is called an element, and each element has a numeric position in an array, known as its index.*

Accessing the Elements of an Array

Array elements can be accessed by their index using the square bracket notation. An index is a number that represents an element's position in an array.

Array indexes are zero-based. This means that the first item of an array is stored at index 0, not 1, the second item is stored at index 1, and so on. Array indexes start at 0 and go up to the number of elements minus 1. So, array of five elements would have indexes from 0 to 4.

The following example will show you how to get individual array element by their index.

```
var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];

document.write(fruits[0]); // Prints: Apple
document.write(fruits[1]); // Prints: Banana
document.write(fruits[2]); // Prints: Mango
document.write(fruits[fruits.length - 1]); // Prints: Papaya
```

Note: *In JavaScript, arrays are really just a special type of objects which has numeric indexes as keys.*
*The **typeof** operator will return "object" for arrays.*

Getting the Length of an Array

The length property returns the length of an array, which is the total number of elements contained in the array. Array length is always greater than the index of any of its element.

```
var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];
document.write(fruits.length); // Prints: 5
```

Looping Through Array Elements

You can use for loop to access each element of an array in sequential order, like this:

```
var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];

// Iterates over array elements
for(var i = 0; i < fruits.length; i++) {
    document.write(fruits[i] + "<br>"); // Print array element
}
```

ECMAScript 6 has introduced a simpler way to iterate over array element, which is for-of loop. In this loop you don't have to initialize and keep track of the loop counter variable (i).

Here's the same example rewritten using the for-of loop:

```
var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];

// Iterates over array elements
for(var fruit of fruits) {
    document.write(fruit + "<br>"); // Print array element
}
```

You can also iterate over the array elements using for-in loop, like this:

```
var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];

// Loop through all the elements in the array
for(var i in fruits) {
    document.write(fruits[i] + "<br>");
}
```

Note: *The for-in loop should not be used to iterate over an array where the index order is important. The for-in loop is optimized for iterating over object's properties, you should better use a for loop with a numeric index or for-of loop.*

Adding New Elements to an Array

To add a new element at the end of an array, simply use the push() method, like this:

```
var colors = ["Red", "Green", "Blue"];
colors.push("Yellow");

document.write(colors); // Prints: Red,Green,Blue,Yellow
document.write(colors.length); // Prints: 4
```

Similarly, to add a new element at the beginning of an array use the unshift() method, like this:

```
var colors = ["Red", "Green", "Blue"];
colors.unshift("Yellow");

document.write(colors); // Prints: Yellow,Red,Green,Blue
document.write(colors.length); // Prints: 4
```

Removing Elements from an Array

To remove the last element from an array you can use the pop() method. This method returns the value that was popped out. Here's an example:

```
var colors = ["Red", "Green", "Blue"];
var last = colors.pop();

document.write(last); // Prints: Blue
document.write(colors.length); // Prints: 2
```

Similarly, you can remove the first element from an array using the shift() method. This method also returns the value that was shifted out. Here's an example:

```
var colors = ["Red", "Green", "Blue"];
var first = colors.shift();

document.write(first); // Prints: Red
document.write(colors.length); // Prints: 2
```

Tip: *The **push()** and **pop()** methods runs faster than **unshift()** and **shift().** Because push() and pop() methods simply add and remove elements at the end of an array therefore the elements do not move, whereas unshift() and shift() add and remove elements at the beginning of the array that require re-indexing of whole array.*

Adding or Removing Elements at Any Position

The splice() method is a very versatile array method that allows you to add or remove elements from any index, using the syntax arr.splice(startIndex, deleteCount, elem1, ..., elemN).

This method takes three parameters: the first parameter is the index at which to start splicing the array, it is required; the second parameter is the number of elements to remove (use 0 if you don't want to remove any elements), it is optional; and the third parameter is a set of replacement elements, it is also optional. The following example shows how it works:

```
var colors = ["Red", "Green", "Blue"];
var removed = colors.splice(0,1); // Remove the first element

document.write(colors); // Prints: Green,Blue
document.write(removed); // Prints: Red (one item array)
document.write(removed.length); // Prints: 1

removed = colors.splice(1, 0, "Pink", "Yellow"); // Insert two items at
position one
document.write(colors); // Prints: Green,Pink,Yellow,Blue
document.write(removed); // Empty array
document.write(removed.length); // Prints: 0

removed = colors.splice(1, 1, "Purple", "Voilet"); // Insert two values,
remove one
document.write(colors); //Prints: Green,Purple,Voilet,Yellow,Blue
document.write(removed); // Prints: Pink (one item array)
document.write(removed.length); // Prints: 1
```

The splice() method returns an array of the deleted elements, or an empty array if no elements were deleted, as you can see in the above example. If the second argument is omitted, all elements from the start to the end of the array are removed. Unlike slice() and concat() methods, the splice() method modifies the array on which it is called on.

Creating a String from an Array

There may be situations where you simply want to create a string by joining the elements of an array. To do this you can use the join() method. This method takes an optional parameter which is a separator string that is added in between each element. If you omit the separator, then JavaScript will use comma (,) by default. The following example shows how it works:

```
var colors = ["Red", "Green", "Blue"];

document.write(colors.join()); // Prints: Red,Green,Blue
document.write(colors.join("")); // Prints: RedGreenBlue
document.write(colors.join("-")); // Prints: Red-Green-Blue
document.write(colors.join(", ")); // Prints: Red, Green, Blue
```

You can also convert an array to a comma-separated string using the toString(). This method does not accept the separator parameter like join(). Here's an example:

```
var colors = ["Red", "Green", "Blue"];
document.write(colors.toString()); // Prints: Red,Green,Blue
```

Extracting a Portion of an Array

If you want to extract out a portion of an array (i.e. subarray) but keep the original array intact you can use the ***slice()*** method. This method takes 2 parameters: start index (index at which to begin extraction), and an optional end index (index before which to end extraction),

```
var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];

document.write(fruits.slice(2)); // Prints: Mango,Orange,Papaya
document.write(fruits.slice(-2)); // Prints: Orange,Papaya
document.write(fruits.slice(2, -1)); // Prints: Mango,Orange
```

like ***arr.slice(startIndex, endIndex)***. Here's an example:

```
var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];
var subarr = fruits.slice(1, 3);
document.write(subarr); // Prints: Banana,Mango
```

If ***endIndex*** parameter is omitted, all elements to the end of the array are extracted. You can also specify negative indexes or offsets —in that case the slice() method extract the elements from the end of an array, rather then the begining. For example:

Merging Two or More Arrays

The ***concat()*** method can be used to merge or combine two or more arrays. This method does not change the existing arrays, instead it returns a new array. For example:

```
var pets = ["Cat", "Dog", "Parrot"];
var wilds = ["Tiger", "Wolf", "Zebra"];

// Creating new array by combining pets and wilds arrays
var animals = pets.concat(wilds);
document.write(animals); // Prints: Cat,Dog,Parrot,Tiger,Wolf,Zebra
```

Searching Through an Array

If you want to search an array for a specific value, you can simply use the ***indexOf()*** and ***lastIndexOf().*** If the value is found, both methods return an index representing the array element. If the value is not found, -1 is returned.
The indexOf() method returns the first one found, whereas the lastIndexOf() returns the last one found.

```
var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];

document.write(fruits.indexOf("Apple")); // Prints: 0
document.write(fruits.indexOf("Banana")); // Prints: 1
document.write(fruits.indexOf("Pineapple")); // Prints: -1
```

Both methods also accept an optional integer parameter from index which specifies the index within the array at which to start the search. Here's an example:

```
var arr = [1, 0, 3, 1, false, 5, 1, 4, 7];

// Searching forwards, starting at from- index
document.write(arr.indexOf(1, 2)); // Prints: 3

// Searching backwards, starting at from index
document.write(arr.lastIndexOf(1, 2)); // Prints: 0
```

Boolean Object

The ***Boolean*** object is a wrapper around the primitive Boolean data type. Use the following syntax to create a ***Boolean*** object:

var booleanObjectName = new Boolean(value);

Do not confuse the primitive Boolean values ***true*** and ***false*** with the true and false values of the Boolean object. Any object whose value is not ***undefined, null, 0, NaN***, or the empty string , including a Boolean object whose value is false, evaluates to true when passed to a conditional statement.

Date Object

JavaScript does not have a date data type. However, you can use the ***Date*** object and its methods to work with dates and times in your applications. The Date object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

JavaScript handles dates similarly to Java. The two languages have many of the same date methods, and both languages store dates as the number of milliseconds since January 1, 1970, 00:00:00.

The Date object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.

To create a Date object:

var dateObjectName = new Date([parameters]);

where ***dateObjectName*** is the name of the ***Date*** object being created; it can be a new object or a property of an existing object.

Calling Date without the new keyword simply converts the provided date to a string representation.

The parameters in the preceding syntax can be any of the following:

- Nothing: creates today's date and time. For example, today = new Date();.

- A string representing a date in the following form: “Month day, year hours:minutes:seconds.” For example, var Xmas95 = new Date("December 25, 1995 13:30:00"). If you omit hours, minutes, or seconds, the value will be set to zero.
- A set of integer values for year, month, and day. For example, var Xmas95 = new Date(1995, 11, 25).
- A set of integer values for year, month, day, hour, minute, and seconds. For example, var Xmas95 = new Date(1995, 11, 25, 9, 30, 0);.

JavaScript 1.2 and earlier

The Date object behaves as follows:

- Dates prior to 1970 are not allowed.
- JavaScript depends on platform-specific date facilities and behavior; the behavior of the Date object varies from platform to platform.

Methods of the Date Object

The Date object methods for handling dates and times fall into these broad categories:

- “set” methods, for setting date and time values in Date objects.
- “get” methods, for getting date and time values from Date objects.
- “to” methods, for returning string values from Date objects.
- parse and UTC methods, for parsing Date strings.

With the “get” and “set” methods you can get and set seconds, minutes, hours, day of the month, day of the week, months, and years separately. There is a ***getDay*** method that returns the day of the week, but no corresponding ***setDay*** method, because the day of the week is set automatically. These methods use integers to represent these values as follows:

- Seconds and minutes: 0 to 59
- Hours: 0 to 23
- Day: 0 (Sunday) to 6 (Saturday)
- Date: 1 to 31 (day of the month)
- Months: 0 (January) to 11 (December)
- Year: years since 1900

For example, suppose you define the following date:

```
var Xmas95 = new Date("December 25, 1995");
```

Then ***Xmas95.getMonth()*** returns 11, and ***Xmas95.getFullYear()*** returns 1995.

The ***getTime*** and ***setTime*** methods are useful for comparing dates. The getTime method returns the number of milliseconds since January 1, 1970, 00:00:00 for a Date object.

For example, the following code displays the number of days left in the current year:

```
var today = new Date();
var endYear = new Date(1995, 11, 31, 23, 59, 59, 999); // Set day and month
endYear.setFullYear(today.getFullYear()); // Set year to this year
var msPerDay = 24 * 60 * 60 * 1000; // Number of milliseconds per day
var daysLeft = (endYear.getTime() - today.getTime()) / msPerDay;
var daysLeft = Math.round(daysLeft); //returns days left in the year
```

This example creates a Date object named today that contains today's date. It then creates a Date object named endYear and sets the year to the current year. Then, using the number of milliseconds per day, it computes the number of days between ***today*** and ***endYear***, using ***getTime*** and rounding to a whole number of days.

The ***parse*** method is useful for assigning values from date strings to existing Date objects. For example, the following code uses parse and ***setTime*** to assign a date value to the ***IPOdate*** object:

```
var IPOdate = new Date();
IPOdate.setTime(Date.parse("Aug 9, 1995"));
```

Using the Date Object: an Example

In the following example, the function ***JSClock()*** returns the time in the format of a digital clock.

```
function JSClock() {
  var time = new Date();
  var hour = time.getHours();
  var minute = time.getMinutes();
  var second = time.getSeconds();
  var temp = "" + ((hour > 12) ? hour - 12 : hour);
  if (hour == 0)
    temp = "12";
  temp += ((minute < 10) ? ":0" : ":") + minute;
  temp += ((second < 10) ? ":0" : ":") + second;
  temp += (hour >= 12) ? " P.M." : " A.M.";
  return temp;
 }
```

The ***JSClock*** function first creates a new Date object called time; since no arguments are given, time is created with the current date and time. Then calls to the getHours, getMinutes, and getSeconds methods assign the value of the current hour, minute and seconds to hour, minute, and second.

The next four statements build a string value based on the time. The first statement creates a variable temp, assigning it a value using a conditional expression; if hour is greater than 12, (hour - 12), otherwise simply hour, unless hour is 0, in which case it becomes 12.

The next statement appends a minute value to temp. If the value of minute is less than 10, the conditional expression adds a string with a preceding zero; otherwise it adds a string with a demarcating colon. Then a statement appends a seconds value to temp in the same way.

Finally, a conditional expression appends “PM” to temp if hour is 12 or greater; otherwise, it appends “AM” to temp.

Function Object

The predefined ***Function*** object specifies a string of JavaScript code to be compiled as a function. To create a Function object:

var functionObjectName = new Function ([arg1, arg2, ... argn], functionBody);

functionObjectName is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as ***window.onerror***.

arg1, arg2, … argn are arguments to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example “x” or "theForm".

functionBody is a string specifying the JavaScript code to be compiled as the function body.

Function objects are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

In addition to defining functions as described here, you can also use the function statement and the function expression. See the JavaScript Reference for more information.

The following code assigns a function to the variable setBGColor. This function sets the current document’s background color.

```
var setBGColor = new Function("document.bgColor = 'antiquewhite'");
```

To call the Function object, you can specify the variable name as if it were a function. The following code executes the function specified by the ***setBGColor*** variable:

```
var colorChoice="antiquewhite";
if (colorChoice=="antiquewhite") {setBGColor()}
```

You can assign the function to an event handler in either of the following ways:

```
document.form1.colorButton.onclick = setBGColor;
```

```
<INPUT NAME="colorButton" TYPE="button"
   VALUE="Change background color"
   onClick="setBGColor()">
```

Creating the variable setBGColor shown above is similar to declaring the following function:

```
function setBGColor() {
  document.bgColor = 'antiquewhite';
}
```

Assigning a function to a variable is similar to declaring a function, but there are differences:

- When you assign a function to a variable using ***var setBGColor = new Function("...")***, setBGColor is a variable for which the current value is a reference to the function created with new Function().
- When you create a function using ***function setBGColor() {...}, setBGColor*** is not a variable, it is the name of a function.

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function:

- The inner function can be accessed only from statements in the outer function.
- The inner function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the inner function.

Math Object

The predefined ***Math*** object has properties and methods for mathematical constants and functions. For example, the Math object's PI property has the value of pi (3.141…), which you would use in an application as

```
Math.PI
```

Similarly, standard mathematical functions are methods of Math. These include trigonometric, logarithmic, exponential, and other functions. For example, if you want to use the trigonometric function sine, you would write

```
Math.sin(1.56)
```

Note *that all trigonometric methods of Math take arguments in radians.*

The following table summarizes the Math object's methods.

Method	Description
abs	Absolute value
sin, cos, tan	Standard trigonometric functions; argument in radians
acos, asin, atan, atan2	Inverse trigonometric functions; return values in radians
exp, log	Exponential and natural logarithm, base e
ceil	Returns least integer greater than or equal to argument
floor	Returns greatest integer less than or equal to argument
min, max	Returns greater or lesser (respectively) of two arguments
pow	Exponential; first argument is base, second is exponent
random	Returns a random number between 0 and 1.
round	Rounds argument to nearest integer
sqrt	Square root

Unlike many other objects, you never create a ***Math*** object of your own. You always use the predefined ***Math*** object.

Number Object

The Number object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

```
var biggestNum = Number.MAX_VALUE;
var smallestNum = Number.MIN_VALUE;
var infiniteNum = Number.POSITIVE_INFINITY;
var negInfiniteNum = Number.NEGATIVE_INFINITY;
var notANum = Number.NaN;
```

You always refer to a property of the predefined Number object as shown above, and not as a property of a Number object you create yourself.

The following table summarizes the Number object's properties.

Property	Description
MAX_VALUE	The largest representable number
MIN_VALUE	The smallest representable number
NaN	Special "not a number" value
NEGATIVE_INFINITY	Special negative infinite value; returned on overflow
POSITIVE_INFINITY	Special positive infinite value; returned on overflow

The Number prototype provides methods for retrieving information from Number objects in various formats. The following table summarizes the methods of ***Number.prototype.***

Method	Description
toExponential	Returns a string representing the number in exponential notation.
toFixed	Returns a string representing the number in fixed-point notation.
toPrecision	Returns a string representing the number to a specified precision in fixed-point notation.
toSource	Returns an object literal representing the specified `Number` object; you can use this value to create a new object. Overrides the `Object.toSource` method.
toString	Returns a string representing the specified object. Overrides the `Object.toString`method.
valueOf	Returns the primitive value of the specified object. Overrides the `Object.valueOf`method.

RegExp Object

The RegExp object lets you work with regular expressions. It is described in RegExp. This will be covered later in the module.

String Object

The String object is a wrapper around the string primitive data type. Do not confuse a string literal with the String object. For example, the following code creates the string literal s1 and also the String object s2:

```
var s1 = "foo"; //creates a string literal value
var s2 = new String("foo"); //creates a String object
```

You can call any of the methods of the String object on a string literal value—JavaScript automatically converts the string literal to a temporary String object, calls the method, then discards the temporary String object. You can also use the ***String.length*** property with a string literal.

You should use string literals unless you specifically need to use a String object, because String objects can have counterintuitive behavior. For example:

```
var s1 = "2 + 2"; //creates a string literal value
var s2 = new String("2 + 2"); //creates a String object
eval(s1); //returns the number 4
eval(s2); //returns the string "2 + 2"
```

A String object has one property, length, that indicates the number of characters in the string. For example, the following code assigns x the value 13, because “Hello, World!” has 13 characters:

```
var mystring = "Hello, World!";
var x = mystring.length;
```

A String object has two types of methods: those that return a variation on the string itself, such as substring and ***toUpperCase***, and those that return an HTML-formatted version of the string, such as bold and link.

For example, using the previous example, both mystring.toUpperCase() and "hello, world!".toUpperCase() return the string “HELLO, WORLD!”

The ***substring*** method takes two arguments and returns a subset of the string between the two arguments. Using the previous example, mystring.substring(4, 9) returns the string "o, Wo". See the substring method of the String object in the JavaScript Reference for more information.

The String object also has a number of methods for automatic HTML formatting, such as bold to create boldface text and link to create a hyperlink. For example, you could create a ***hyperlink*** to a hypothetical URL with the link method as follows:

```
mystring.link("http://www.helloworld.com");
```

The following table summarizes the methods of String objects.

Method	Description
anchor	Creates HTML named anchor.
big, blink, bold, fixed, italics, small, strike, sub, sup	Create HTML formatted string.
charAt, charCodeAt	Return the character or character code at the specified position in string.
indexOf, lastIndexOf	Return the position of specified substring in the string or last position of specified substring, respectively.
link	Creates HTML hyperlink.
concat	Combines the text of two strings and returns a new string.
fromCharCode	Constructs a string from the specified sequence of Unicode values. This is a method of the String class, not a String instance.
split	Splits a `String` object into an array of strings by separating the string into substrings.
slice	Extracts a section of an string and returns a new string.
substring, substr	Return the specified subset of the string, either by specifying the start and end indexes or the start index and a length.
match, replace, search	Work with regular expressions.
toLowerCase, toUpperCase	Return the string in all lowercase or all uppercase, respectively.

Document Object Model

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

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

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

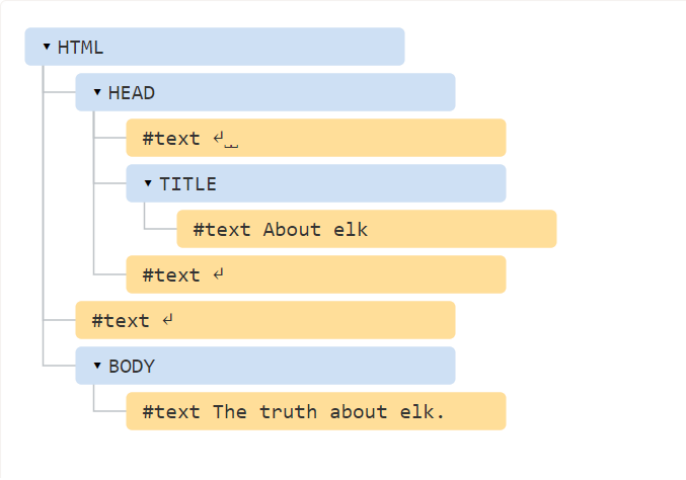
In this section we'll cover the HTML DOM which provides a standard interface for accessing and manipulating HTML documents through JavaScript. With the HTML DOM, you can use JavaScript to build HTML documents, navigate their hierarchical structure, and add, modify, or delete elements and attributes or their content, and so on. Almost anything found in an HTML document can be accessed, changed, deleted, or added using the JavaScript with the help of HTML DOM.

An example of the DOM

Let's start with the following simple document:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>About elk</title>
</head>
<body>
  The truth about elk.
</body>
</html>
```

The DOM represents HTML as a tree structure of tags. Here's how it looks:



On the picture above, you can click on element nodes and their children will open/collapse.
Every tree node is an object.

Tags are ***element nodes*** (or just elements) and form the tree structure: ***<html>*** is at the root, then ***<head>*** and ***<body>*** are its children, etc.
The text inside elements forms text nodes, labelled as **#text**. A text node contains only a string. It may not have children and is always a leaf of the tree.

For instance, the ***<title>*** tag has the text ***"About elk"***.

Please note the special characters in text nodes:

- a newline: ↲ (in JavaScript known as \n)
- a space: ␣

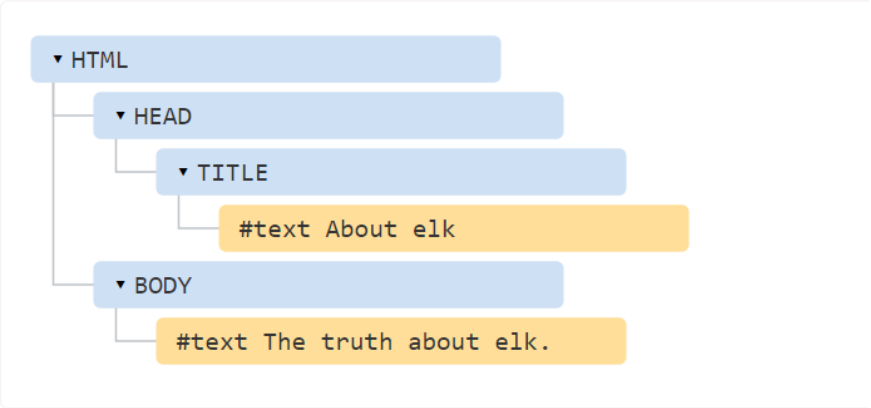
Spaces and newlines are totally valid characters, like letters and digits. They form text nodes and become a part of the DOM. So, for instance, in the example above the ***<head>*** tag contains some spaces before ***<title>***, and that text becomes a ***#text*** node (it contains a newline and some spaces only).

There are only two top-level exclusions:

- Spaces and newlines before ***<head>*** are ignored for historical reasons.
- If we put something after ***</body>,*** then that is automatically moved inside the body, at the end, as the HTML spec requires that all content must be inside <body>. So, there can't be any spaces after </body>.

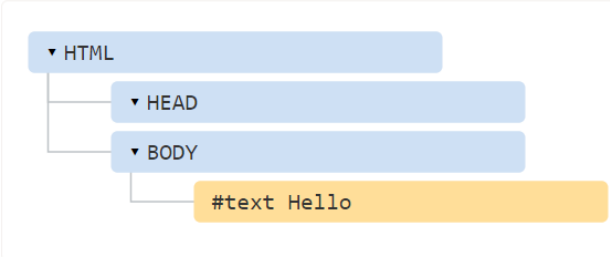
In other cases everything's straightforward – if there are spaces (just like any character) in the document, then they become text nodes in the DOM, and if we remove them, then there won't be any.
Here are no space-only text nodes:

```
<!DOCTYPE HTML>
<html><head><title>About elk</title></head><body>The truth about elk.</body></html>
```



Autocorrection

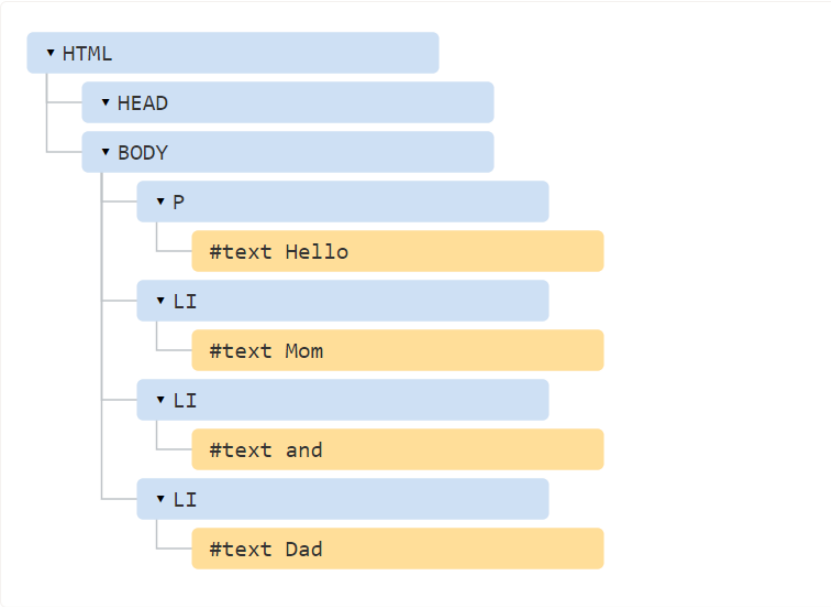
If the browser encounters malformed HTML, it automatically corrects it when making the DOM.
For instance, the top tag is always <html>. Even if it doesn't exist in the document, it will exist in the DOM, because the browser will create it. The same goes for <body>.
As an example, if the HTML file is the single word "Hello", the browser will wrap it into <html> and <body>, and add the required <head>, and the DOM will be:



While generating the DOM, browsers automatically process errors in the document, close tags and so on.
A document with unclosed tags:

```
<p>Hello
<li>Mom
<li>and
<li>Dad
```

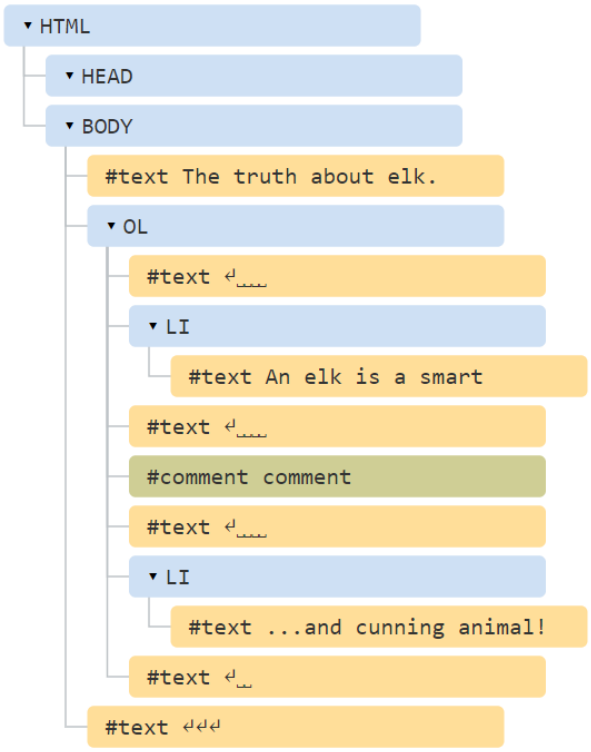
…will become a normal DOM as the browser reads tags and restores the missing parts:



Other Node Types:

There are some other node types besides elements and text nodes.
For example, comments:

```
<!DOCTYPE HTML>
<html>
<body>
  The truth about elk.
  <ol>
    <li>An elk is a smart</li>
    <!-- comment -->
    <li>...and cunning animal!</li>
  </ol>
</body>
</html>
```



We can see here a new tree node type – comment node, labeled as #comment, between two text nodes.

We may think – why is a comment added to the DOM? It doesn't affect the visual representation in any way. But there's a rule – if something's in HTML, then it also must be in the DOM tree.

Everything in HTML, even comments, becomes a part of the DOM.

Even the <!DOCTYPE...> directive at the very beginning of HTML is also a DOM node. It's in the DOM tree right before <html>. Few people know about that. We are not going to touch that node, we even don't draw it on diagrams, but it's there.

The document object that represents the whole document is, formally, a DOM node as well.

There are 12 node types. In practice we usually work with 4 of them:

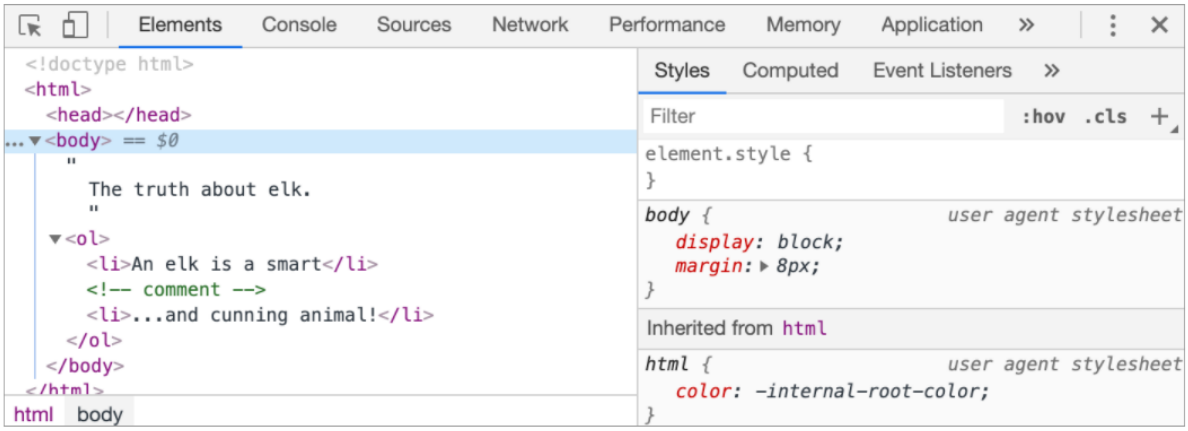
- document – the "entry point" into DOM.
- element nodes – HTML-tags, the tree building blocks.
- text nodes – contain text.
- comments – sometimes we can put information there, it won't be shown, but JS can read it from the DOM.

Working with DOM Structure In Real-time

To see the DOM structure in real-time, try Live DOM Viewer (http://software.hixie.ch/utilities/js/live-dom-viewer/). Just type in the document, and it will show up as a DOM at an instant.

Another way to explore the DOM is to use the browser developer tools. Actually, that's what we use when developing.

To do so, open the web page elk.html (Above mentioned example saved in elk.html), turn on the browser developer tools and switch to the Elements tab. It should look like this:

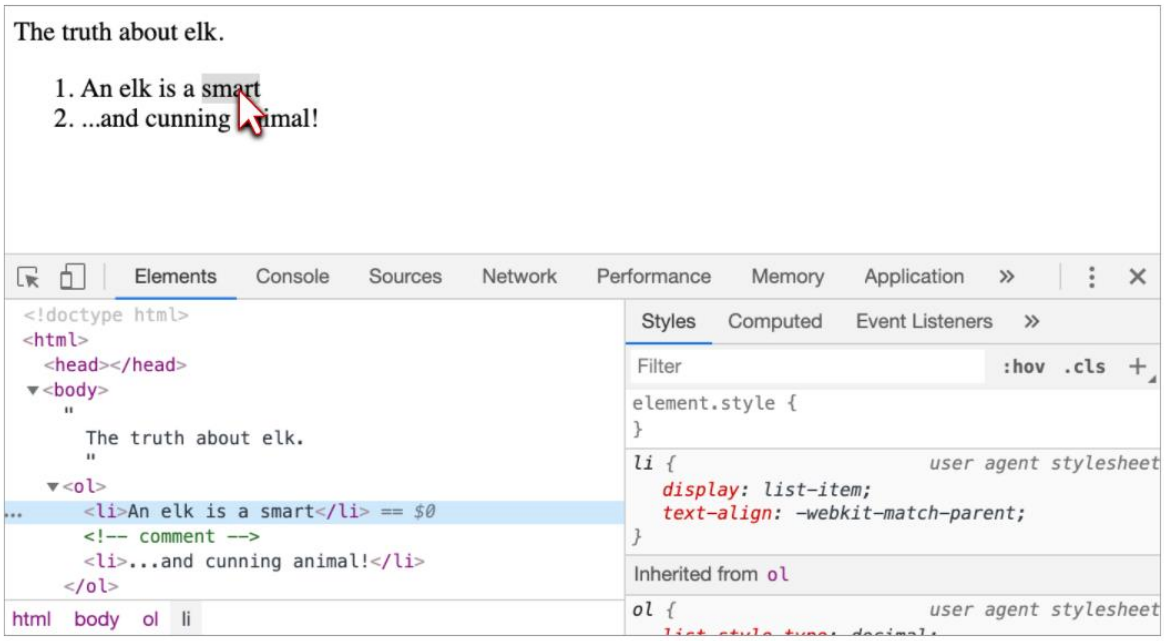


You can see the DOM, click on elements, see their details and so on.

Please note that the DOM structure in developer tools is simplified. Text nodes are shown just as text. And there are no “blank” (space only) text nodes at all. That’s fine, because most of the time we are interested in element nodes.

Clicking the button in the left-upper corner allows us to choose a node from the webpage using a mouse (or other pointer devices) and “inspect” it (scroll to it in the Elements tab). This works great when we have a huge HTML page (and corresponding huge DOM) and would like to see the place of a particular element in it.

Another way to do it would be just right-clicking on a webpage and selecting “Inspect” in the context menu.



At the right part of the tools there are the following subtabs:

- Styles – we can see CSS applied to the current element rule by rule, including built-in rules (gray). Almost everything can be edited in-place, including the dimensions/margins/paddings of the box below.
- Computed – to see CSS applied to the element by property: for each property we can see a rule that gives it (including CSS inheritance and such).
- Event Listeners – to see event listeners attached to DOM elements (we'll cover them in the next part of the tutorial).
- …and so on.

The best way to study them is to click around. Most values are editable in-place.

Interaction with console

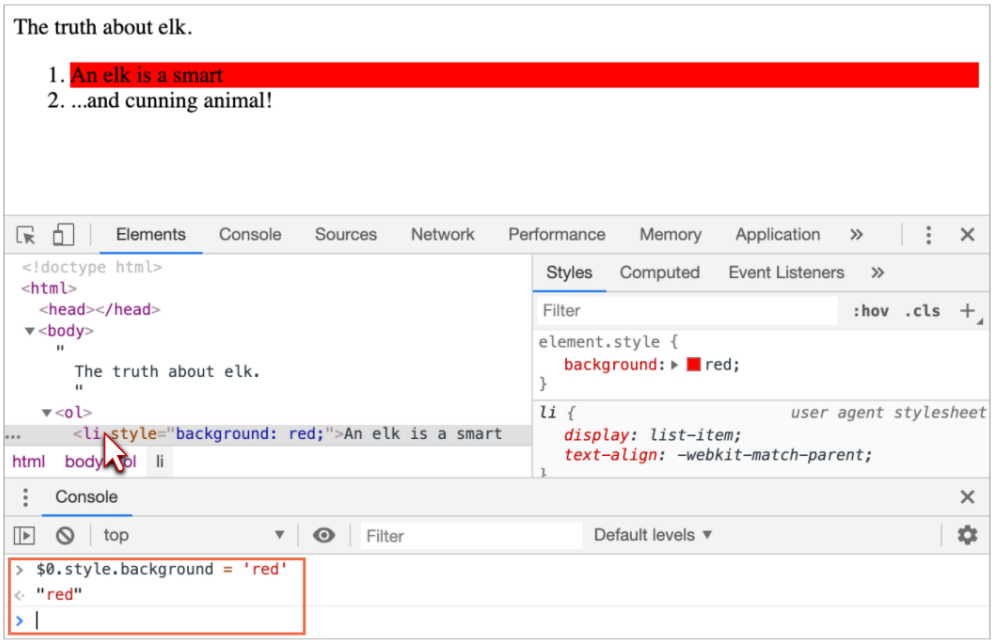
As we work the DOM, we also may want to apply JavaScript to it. Like: get a node and run some code to modify it, to see the result. Here are few tips to travel between the Elements tab and the console.

For the start:

- Select the first <li> in the Elements tab.
- Press Esc – it will open console right below the Elements tab.

Now the last selected element is available as ***$0***, the previously selected is ***$1*** etc.

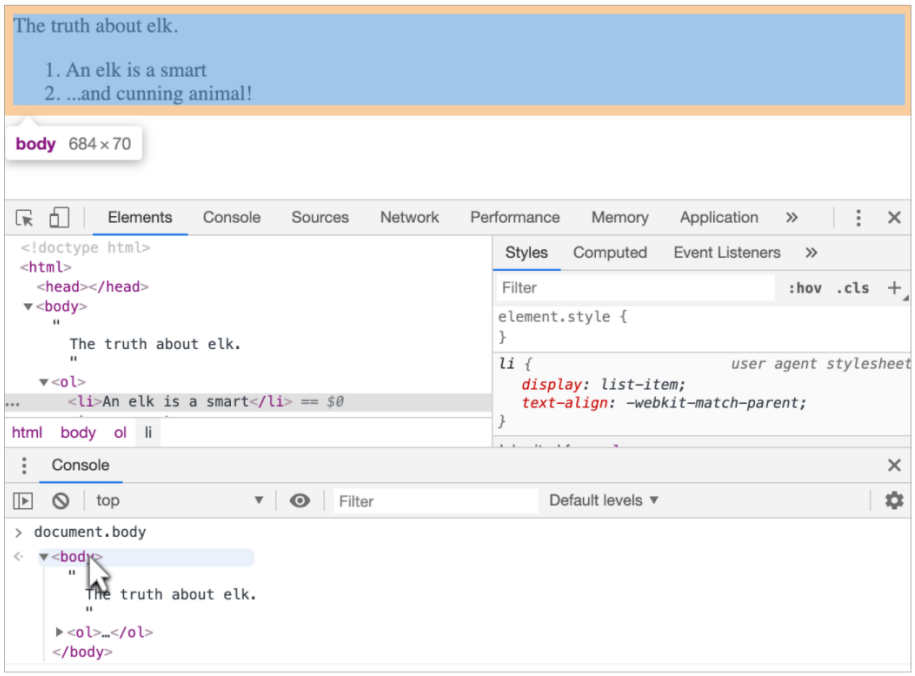
We can run commands on them. For instance, ***$0.style.background = 'red'*** makes the selected list item red, like this:



that's how to get a node from Elements in Console.

There's also a road back. If there's a variable referencing a DOM node, then we can use the command ***inspect(node)*** in Console to see it in the Elements pane.

Or we can just output the DOM node in the console and explore "in-place", like ***document.body*** below:



That's for debugging purposes of course. From the next chapter on we'll access and modify DOM using JavaScript.

The browser developer tools are a great help in development: we can explore the DOM, try things and see what goes wrong.

Summary

An HTML/XML document is represented inside the browser as the DOM tree.

- Tags become element nodes and form the structure.
- Text becomes text nodes.
- …etc, everything in HTML has its place in DOM, even comments.

We can use developer tools to inspect DOM and modify it manually.

Here we covered the basics, the most used and important actions to start with. There's an extensive documentation about Chrome Developer Tools at https://developers.google.com/web/tools/chrome-devtools . The best way to learn

the tools is to click here and there, read menus: most options are obvious. Later, when you know them in general, read the docs and pick up the rest.

DOM nodes have properties and methods that allow us to travel between them, modify them, move around the page, and more.

JavaScript Regular Expressions

What is Regular Expression

Regular Expressions, commonly known as "***regex***" or "***RegExp***", are a specially formatted text strings used to find patterns in text. Regular expressions are one of the most powerful tools available today for effective and efficient text processing and manipulations. For example, it can be used to verify whether the format of data i.e. name, email, phone number, etc. entered by the user is correct or not, find or replace matching string within text content, and so on.

JavaScript supports Perl style regular expressions. Why Perl style regular expressions? Because Perl (Practical Extraction and Report Language) was the first mainstream programming language that provided integrated support for regular expressions and it is well known for its strong support of regular expressions and its extraordinary text processing and manipulation capabilities.

Let's begin with a brief overview of the commonly used JavaScript's built-in methods for performing pattern-matching before delving deep into the world of regular expressions.

Function	What it Does
`exec()`	Search for a match in a string. It returns an array of information or `null` on mismatch.
`test()`	Test whether a string matches a pattern. It returns `true` or `false`.
`search()`	Search for a match within a string. It returns the index of the first match, or `-1` if not found.
`replace()`	Search for a match in a string, and replaces the matched substring with a replacement string.
`match()`	Search for a match in a string. It returns an array of information or `null` on mismatch.
`split()`	Splits up a string into an array of substrings using a regular expression.

Note: *The methods exec() and test() are RegExp methods that takes a string as a parameter, whereas the methods search(), replace(), match() and split() are String methods that takes a regular expression as a parameter.*

Defining Regular Expressions

In JavaScript, regular expressions are represented by RegExp object, which is a native JavaScript object like String, Array, and so on. There are two ways of creating a new RegExp object — one is using the literal syntax, and the other is using the ***RegExp()*** constructor.

The literal syntax uses forward slashes ***(/pattern/)*** to wrap the regular expression pattern, whereas the constructor syntax uses quotes ***("pattern").*** The following example demonstrates both ways of creating a regular expression that matches any string that begins with "Mr.".

```
// Literal syntax
var regex = /^Mr\./;

// Constructor syntax
var regex = new RegExp("^Mr\\.");
```

As you can see, the regular expression literal syntax is shorter and easier to read. Therefore, it is preferable to use the literal syntax.

Note: *When using the constructor syntax, you've to double-escape special characters, which means to match "." you need to write "\\." instead of "\.". If there is only one backslash, it would be interpreted by JavaScript's string parser as an escaping character and removed.*

Pattern Matching with Regular Expression

Regular expression patterns include the use of letters, digits, punctuation marks, etc., plus a set of special regular expression characters (do not confuse with the HTML special characters).

The characters that are given special meaning within a regular expression, are:

. * ? + [ ] ( ) { } ^ $ | \. You will need to backslash these characters whenever you want to use them literally. For example, if you want to match ".", you'd have to write **\.**

All other characters automatically assume their literal meanings.

Character Classes

Square brackets surrounding a pattern of characters are called a character class e.g. ***[abc]***. A character class always matches a single character out of a list of specified characters that means the expression ***[abc]*** matches only a, b or c character.

Negated character classes can also be defined that match any character except those contained within the brackets. A negated character class is defined by placing a caret ***(^)*** symbol immediately after the opening bracket, like ***[^abc],*** which matches any character except a, b, and c.

You can also define a range of characters by using the hyphen (-) character inside a character class, like [0-9]. Let's look at some examples of the character classes:

RegExp	What it Does
`[abc]`	Matches any one of the characters a, b, or c.
`[^abc]`	Matches any one character other than a, b, or c.
`[a-z]`	Matches any one character from lowercase a to lowercase z.
`[A-Z]`	Matches any one character from uppercase a to uppercase z.
`[a-Z]`	Matches any one character from lowercase a to uppercase Z.
`[0-9]`	Matches a single digit between 0 and 9.
`[a-z0-9]`	Matches a single character between a and z or between 0 and 9.

The following example will show you how to find whether a pattern exists within a string or not using the regular expression with the JavaScript ***test()*** method:

```
var regex = /ca[kf]e/;
var str = "He was eating cake in the cafe.";

// Test the string against the regular expression
if(regex.test(str)) {
    alert("Match found!");
} else {
    alert("Match not found.");
}
```

Tip: *Regular expressions aren't exclusive to JavaScript. Languages such as Java, Perl, Python, PHP, etc. use the same notation for finding patterns in text.*

Exercise sixteen:

1. Create a new JavaScript file.

2. Using all of the techniques, concepts and code that you have learned. The point of this final exercise is to familiarize with JavaScript.

ES6

ECMAScript 6 was the second major revision to JavaScript. ECMAScript 6 is also known as ES6 and ECMAScript 2015.

ECMAScript (ES) is a scripting language specification standardized by ECMAScript International. Languages like JavaScript, Jscript and ActionScript are governed by this specification.

ECMASCRIPT was created to standardize JavaScript.

Edition	Date published	Changes from prior edition
ES 1	June 1997	First edition
ES 2	June 1998	Editorial changes
ES 3	December 1999	Added regular expressions
ES 4	*Abandoned*	Fourth Edition was abandoned,
ES 5	December 2009	Adds "strict mode," Clarifies many ambiguities in the 3rd edition specification, Adds some new features, such as getters and setters,
ES 6	June 2015	Classes and modules, iterators and for/of loops, generators, arrow functions, typed arrays, collections (maps, sets and weak maps), promises,

Features of ES6:

1. Var, Let and Const keyword
2. Arrow functions, default arguments
3. Template Strings, String methods
4. Object Destructuring
5. Spread and Rest operator

Block Scoping with *let*:

Before the advent of ES6, var declarations ruled as King.
Var declarations are globally scoped or function/locally scoped

```
var greeter = "hey hi";
function newFunction() {
var hello = "hello";
}
```

Here, greeter is globally scoped because it exists outside a function while hello is function scoped. So we cannot access

the variable hello outside of a function. let is preferred for variable declaration now.

A block is chunk of code bounded by {}. A block lives in curly braces. Anything within curly braces is a block. So a variable declared in a block with the let is only available for use within that block

```
let greeting = "say Hi";
let times = 4;
if (times > 3) {
let hello = "say Hello instead";
console.log(hello);//"say Hello instead"
}
console.log(hello) // hello is not defined
```

We see that using hello outside its block (the curly braces where it was defined) returns an error. This is because let variables are block scoped.

Const Keyword:

Variables declared with the const maintain constant values. const declarations share some similarities with let declarations. Like let declarations, const declarations can only be accessed within the block it was declared.

```
const greeting = "say Hi";
greeting = "say Hello instead"; //error : Assignment to constant variable.
```

Arrow Functions:

An arrow function expression has a shorter syntax than a function expression and does not have its own this, arguments, super, or new.target.
These function expressions are best suited for non-method functions, and they cannot be used as constructors.
It is an anonymous function expression that points to a single line of code. Following is the syntax for the same.

```
([param1, parma2,…param n] )=>statement;
```

The syntax for arrow functions comes in many flavours depending upon what you're trying to accomplish. All variations begin with function arguments, followed by the arrow, followed by the body of the function. Both the arguments and the body can take different forms depending on usage. For example, the following arrow function takes a single argument and simply returns it:

```
var reflect = value => value;
// effectively equivalent to:
var reflect = function(value) {
return value;
}
```

If you are passing in more than one argument, then you must include parentheses around those arguments, like this:

```
var sum = (num1, num2) => num1 + num2;
// effectively equivalent to:
var sum = function(num1, num2) {
return num1 + num2;
};
```

Default function parameters:

In ES6, a function allows the parameters to be initialized with default values, if no values are passed to it or it is undefined. The same is illustrated in the following code:

```
function add(a, b = 1) {
return a+b;
}
console.log(add(4))
```

The above function, sets the value of b to 1 by default. The function will always consider the parameter b to bear the value 1 unless a value has been explicitly passed.

Spread operator / Rest Parameters:

ES6 introduced "…" operator which is also called as spread operator. When "…" operator is applied on an array it expands the array into multiple variables in syntax wise. When its applied to a function argument it makes the function argument behave like array of arguments.
To declare a rest parameter, the parameter name is prefixed with three periods, known as the spread operator.

```
function fun1(...params) {
console.log(params.length);
}
fun1();
fun1(5);
fun1(5, 6, 7);
```

Template Strings, String methods:

Syntactically these are strings that use backticks (i.e. `) instead of single (') or double (") quotes. The motivation of Template Strings is three fold:

- String Interpolation
- Multiline Strings
- Tagged Templates

String interpolation:

A common use case is when you want to generate some string out of some static strings + some variables. For this you would need some templating logic and this is where template strings get their name from. Here's how you would potentially generate an html string previously:

```
var lyrics = 'Never gonna give you up';
var html = '<div>' + lyrics + '</div>';
Now with template strings you can just do:
var lyrics = 'Never gonna give you up';
var html = `<div>${lyrics}</div>`;
```

Multiple String:

Ever wanted to put a newline in a JavaScript string? Perhaps you wanted to embed some lyrics? You would have needed to escape the literal newline using our favorite escape character \, and then put a new line into the string manually \n at the next line. This is shown below:

```
var lyrics = "Never gonna give you up \\n Never gonna let you down";
```

With TypeScript you can just use a template string:

```
var lyrics = `Never gonna give you up
Never gonna let you down`;
```

Tagged Templates:

You can place a function (called a tag) before the template string and it gets the opportunity to pre process the template string literals plus the values of all the placeholder expressions and return a result.
A few notes:

- All the static literals are passed in as an array for the first argument

Here is an example where we have a tag function (named htmlEscape) that escapes the html from all the placeholders:

```
var say = "a bird in hand > two in the
bush";
var html = htmlEscape `<div> I would just
like to say : ${say}</div>`;
```

Object de-structuring:

Destructuring is a way of extracting values into variables from data stored in objects and arrays.Let's imagine we have an object like so:

```
const obj = {first: 'Asim', last: 'Hussain', age: 39 };
```

We want to extract the first and last properties into local variables, prior to ES6 we would have to write something like this:

```
const f = obj.first;
const l = obj.last;
console.log(f); // Asim
console.log(l); // Hussain
```

With destructing we can do so in one line, like so:

```
const {first: f, last: l} = obj;
console.log(f); // Asim
console.log(l); // Hussain
```

Typescript is a strongly typed, object oriented, compiled language. Typescript is a typed superset of JavaScript compiled to JavaScript. In other words, Typescript is JavaScript plus some additional features.
Angular is built in a JavaScript-like language called Typescript.
Typescript isn't a completely new language, it's a superset of ES6. If we write ES6 code, it's perfectly valid and compliable Typescript code.

Typescript code is written in .ts files, which can't be used directly in the browser and need to be translated to vanilla .js first. This compilation process can be done in a number of different ways:

– In the terminal using the previously mentioned command line tool tsc.
– Directly in Visual Studio or some of the other IDEs and text editors.
– Using automated task runners such as gulp.

There are five big improvements that Typescript bring over ES6:

- Types
- Classes
- Decorators
- Imports
- language utilities (e.g. destructuring)

The major improvement of TypeScript over ES6, that gives the language its name, is the typing system. For some people the lack of type checking is considered one of the benefits of using a language like JavaScript.

- It Helps When Writing Code Because It Can Prevent Bugs at Compile Time And
- It helps when reading code because it clarifies your intentions

Basic types:

Boolean

The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a boolean value.

```
let isDone: boolean = false;
```

Number

As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type number. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

String

Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type string to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (") or single quotes (') to surround string data.

```
let color: string = "blue";
color = 'red';
```

You can also use template strings, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }.

Array

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by [] to denote an array of that element type:

```
let list: number[] = [1, 2, 3];
```

The second way uses a generic array type, Array<elemType>:

```
let list: Array<number> = [1, 2, 3];
```

Tuple

Tuple types allow you to express an array where the type of a fixed number of elements is known, but need not be the same. For example, you may want to represent a value as a pair of a string and a number:

```
// Declare a tuple type
let x: [string, number];
```

```
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

When accessing an element with a known index, the correct type is retrieved:

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```

Enum

A helpful addition to the standard set of datatypes from JavaScript is the enum. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0:

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

Any

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks. To do so, we label these with the any type:

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a Boolean
```

The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. You might expect Object to play a similar role, as it does in other languages. But variables of type Object only allow you to assign any value to them - you can't call arbitrary methods on them, even ones that actually exist:

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)
let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

The any type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

```
let list: any[] = [1, true, "free"];
list[1] = 100;
```

Void

void is a little like the opposite of any: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {
console.log("This is my warning message");
}
```

Declaring variables of type void is not useful because you can only assign undefined or null to them:

```
let unusable: void = undefined;
```

Null and Undefined

In TypeScript, both undefined and null actually have their own types named undefined and null respectively. Much like void, they're not extremely useful on their own:

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

By default null and undefined are subtypes of all other types. That means you can assign null and undefined to something like number.

Never

The never type represents the type of values that never occur. For instance, never is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns; Variables also acquire the type never when narrowed by any type guards that can never be true.
The never type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to, never (except never itself). Even any isn't assignable to never.

Type assertions

Sometimes you'll end up in a situation where you'll know more about a value than TypeScriptdoes. Usually this will happen when you know the type of some entity could be more specific than its current type.
Type assertions are a way to tell the compiler "trust me, I know what I'm doing." A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data. It has no runtime impact, and is used purely by the compiler. TypeScriptassumes that you, the programmer, have performed any special checks that you need.
Type assertions have two forms. One is the "angle-bracket" syntax:

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
```

Classes

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components,
Classes inherit functionality and objects are built from these classes.
In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript/

Let's take a look at a simple class-based example:

```
class Greeter {
greeting: string;
constructor(message: string) {
this.greeting = message;
  }
greet() {
return "Hello, " + this.greeting;
```

```
  }
}
let greeter = new Greeter("world");
```

Inheritance

In TypeScript, we can use common object-oriented patterns. One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

```
class Animal {
move(distanceInMeters: number = 0) {
console.log(`Animal moved ${distanceInMeters}m.`);
    }
}
class Dog extends Animal {
bark() {
console.log('Woof! Woof!');
  }
}
const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

Public, private, and protected modifiers

- Public by default
  - In our examples, we've been able to freely access the members that we declared throughout our programs
  - In TypeScript, each member is public by default.

- You may still mark a member public explicitly. We could have written the Animal

  class from the previous section in the following way:

```
class Animal {
public name: string;
```

```
public constructor(theName: string) { this.name = theName; }
public move(distanceInMeters: number) {
console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

Readonly modifier

You can make properties readonly by using the readonly keyword. Readonly properties must be initialized at their declaration or in the constructor.

```
class Octopus {
readonly name: string;
readonly numberOfLegs: number = 8;
constructor (theName: string) {
this.name = theName;
    }
}
let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // error! name is readonly.
```

Angular

What is Angular6?

1. Open Source JavaScript framework used to build web applications
2. Developed by Google
3. Excellent framework for building single phase applications and line of business applications

Advantages of angular 6?

1. Dependency Injection
2. Two way data binding
3. Testing
4. Model View Controller

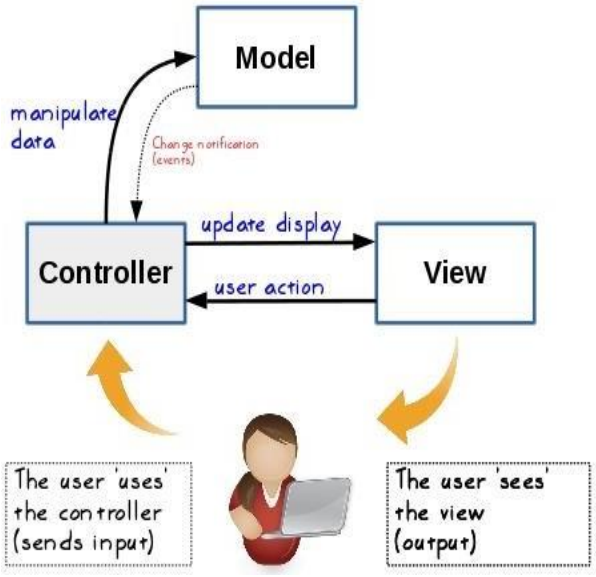
Angular 6 MVC

MVC, which stands for **M**odel **V**iew and **C**ontroller is a software design pattern for developing web applications. It is popular because it separates the application logic from the user interface layer and supports separation of concerns.

Model: It is the lowest level of the pattern which is responsible for maintaining the data

View: It is responsible for displaying all or part of data to the users. t also specifies the data in a particular format triggered by the controller's decision to present the data

Controller: It controls the relation between models and views. When the user sends a request, the controller interacts with



Example:

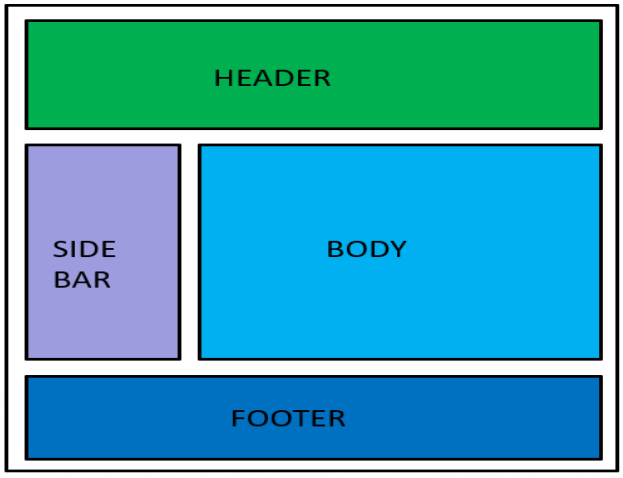
```
<!DOCTYPE html>
<html>
<head>
<script  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.5/angular.min.js"></script>
</head>
<body>
.
.
</body>
</html>
```

Component based model

- The traditional approach of front end web development involves the following:
    - **HTML** – Static portion of application
    - **JavaScript** – Dynamic portion of application

- Since the JavaScript is embedded into the HTML file, it loads along with the HTML page.
- This allows JavaScript to manipulate DOM and achieve dynamic functionality. With Angular 6, our approach changes. We now use component-based approach. As Angular developers, we will thinkof the components that we need to design, create, build and put together so that we have the application.
- Let's understand component-based model with the help of an example. In most of the websites, we find a header, a body, the side bar and the footer.
- We need to try to figure out the different options in this application that you can separate as components. These components are self sufficient and knows what to with that area. In our case,all these elements will be our components.
- Each of the component will have HTML and JavaScript that can be combined together. These components are used by creating an HTML tag or selector for it.
- This can be done by assigning a name or selector to the component. The consumer uses these

selectors to call and render the components.



Let's say we have a selector called header-section. This selector can be used as a tag in any HTML in yourAngular code and when someone uses the tag, the header is rendered.

These components can have sub-components, forming a component tree.

Every angular component has a root component which holds the main components that are going to be displayed in the page.

While building angular application, you need to think in terms of components.

Setting up environment

In order to build an application in Angular, we need to install 3 things:

1. Nodejs - Runtime environment for Angular – Go to nodejs.org and download the installer
2. Editor - We are going to visual studio code. Go to code.visualstudio.com and download the installer

3. Angular CLI - A way in which you can start up a simple angular project without having to assemble all the files and start everything yourself. Go to cli.angular.io and download the CLI

Angular 6 components

```
angular. module('myApp').

component('greetUser', { template: 'Hello, {{$ctrl.user}}!',

controller: function GreetUserController() { this.user = 'world';

}

});

<body>

<!-- The following line is how to use the `greetUser` component above in your html doc. -->

<greet-user></greet-user>

</body>
```

Angular 6 directives

- AngularJS lets you extend HTML with new attributes called Directives.
- AngularJS has a set of built-in directives which offers functionality to your applications.
- AngularJS also lets you define your own directives.

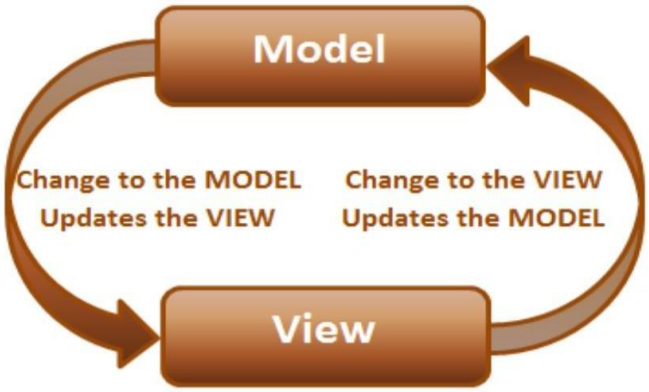
Angular 6 has three directives. Let's list them down!

1. Component Directives: Directives with a template
2. Structural Directives: Change the DOM layout by adding and removing DOM elements
3. Attribute Directives: Change the appearance or behavior of an element, component or another directive

Angular 6 Data Binding

Data-binding is the process of synchronization of data between a model and a view. Angular follow three types of Data-binding method:

1. One-way Data-binding
2. Two-way Data-binding
3. One-Time Data-binding



ANGULAR 6 services

- Just as its name indicates, a service is a function or object.
- These services are available for and limited to your AngularJS application
- AngularJS has about 30 built-in services.
- Let's have a look at some of the most common AngularJS services

React JS

ReactJS is JavaScript library used for building reusable UI components. According to React official documentation, following is the definition −

React is a library for building composable user interfaces. It encourages the creation of reusable UI components, which present data that changes over time. Lots of people use React as the V in MVC. React abstracts away the DOM from you, offering a simpler programming model and better performance. React can also render on the server using Node, and it can power native apps using React Native. React implements one-way reactive data flow, which reduces the boilerplate and is easier to reason about than traditional data binding.

How does React works?

React creates a VIRTUAL DOM in memory.

Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.

React only changes what needs to be changed!

React finds out what changes have been made, and changes only what needs to be changed.

Environment Setup

First need to install Node.js before starting with React.

After successfully installing NodeJS, we can start installing React upon it using npm. You can install ReactJS in two ways

- Using webpack and babel.
- Using the create-react-app command.

Step 1 - Create the Root Folder

Create a folder with name **reactApp** on the desktop to install all the required files, using the mkdir command.

```
C:\Users\username\Desktop>mkdir reactApp
```

```
C:\Users\username\Desktop>cd reactApp
```

To create any module, it is required to generate the **package.json** file. Therefore, after Creating the folder, we need to create a **package.json** file. To do so you need to run the **npm init** command from the command prompt.

```
C:\Users\username\Desktop\reactApp>npm init
```

This command asks information about the module such as packagename, description, author etc. you can skip these using the –y option.

```
C:\Users\username\Desktop\reactApp>npm init -y
```

Wrote to C:\reactApp\package.json:

```
{
  "name": "reactApp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Step 2 - install React and react dom

Since our main task is to install ReactJS, install it, and its dom packages, using **install react** and **react-dom** commands of npm respectively. You can add the packages we install, to **package.json** file using the **--save** option.

```
C:\Users\Tutorialspoint\Desktop\reactApp>npm install react --save
C:\Users\Tutorialspoint\Desktop\reactApp>npm install react-dom --save
Or, you can install all of them in single command as –
C:\Users\username\Desktop\reactApp>npm install react react-dom –save
```

Step 3 - Install webpack

Since we are using webpack to generate bundler install webpack, webpack-dev-server and webpack-cli.

```
C:\Users\username\Desktop\reactApp>npm install webpack --save
C:\Users\username\Desktop\reactApp>npm install webpack-dev-server --save
C:\Users\username\Desktop\reactApp>npm install webpack-cli --save
Or, you can install all of them in single command as −
C:\Users\username\Desktop\reactApp>npm install webpack webpack-dev-server webpack-cli –save
```

Step 4 - Install babel

Install babel, and its plugins babel-core, babel-loader, babel-preset-env, babel-preset-react and, html-webpack-plugin

```
C:\Users\username\Desktop\reactApp>npm install babel-core --save-dev
C:\Users\username\Desktop\reactApp>npm install babel-loader --save-dev
C:\Users\username\Desktop\reactApp>npm install babel-preset-env --save-dev
C:\Users\username\Desktop\reactApp>npm install babel-preset-react --save-dev
C:\Users\username\Desktop\reactApp>npm install html-webpack-plugin --save-dev
Or, you can install all of them in single command as −
C:\Users\username\Desktop\reactApp>npm install babel-core babel-loader babel-preset-env
  babel-preset-react html-webpack-plugin --save-dev
```

Step 5 - Create the Files

To complete the installation, we need to create certain files namely, index.html, App.js, main.js, webpack.config.js and, .***babelrc***. You can create these files manually or, using **command prompt**.

```
C:\Users\username\Desktop\reactApp>type nul > index.html
C:\Users\username\Desktop\reactApp>type nul > App.js
C:\Users\username\Desktop\reactApp>type nul > main.js
C:\Users\username\Desktop\reactApp>type nul > webpack.config.js
C:\Users\username\Desktop\reactApp>type nul > .babelrc
```

Step 6 - Set Compiler, Server and Loaders

Open **webpack-config.js** file and add the following code. We are setting webpack entry point to be main.js. Output path is the place where bundled app will be served. We are also setting the development server to **8001** port. You can choose any port you want.

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './main.js',
  output: {
    path: path.join(__dirname, '/bundle'),
    filename: 'index_bundle.js'
  },
  devServer: {
    inline: true,
    port: 8001
  },
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  },
  plugins:[
    new HtmlWebpackPlugin({
      template: './index.html'
    })
  ]
}
```

Open the **package.json** and delete **"test" "echo \"Error: no test specified\" && exit 1"** inside **"scripts"** object. We are deleting this line since we will not do any testing in this tutorial. Let's add the **start** and **build** commands instead.

"start": "webpack-dev-server --mode development --open --hot",

"build": "webpack --mode production"

Step 7 - index.html

This is just regular HTML. We are setting **div id = "app"** as a root element for our app and adding **index_bundle.js** script, which is our bundled app file.

```
<!DOCTYPE html>
<html lang = "en">
  <head>
    <meta charset = "UTF-8">
    <title>React App</title>
  </head>
  <body>
    <div id = "app"></div>
    <script src = 'index_bundle.js'></script>
  </body>
</html>
```

Step 8 − App.jsx and main.js

This is the first React component. We will explain React components in depth in a subsequent chapter. This component will render **Hello World**.

App.js

```
import React, { Component } from 'react';
class App extends Component{
  render(){
    return(
      <div>
        <h1>Hello World</h1>
      </div>
    );
  }
}
export default App;
```

We need to import this component and render it to our root **App** element, so we can see it in the browser.

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

ReactDOM.render(<App />, document.getElementById('app'));
```

```
{
  "presets":["env", "react"]
}
```

Step 9 - Running the Server

The setup is complete and we can start the server by running the following command.

```
C:\Users\username\Desktop\reactApp>npm start
```

Step 10 - Generating the bundle

Finally, to generate the bundle you need to run the build command in the command prompt as −

```
C:\Users\Desktop\reactApp>npm run build
```

React Lifecycle

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.
The three phases are: **Mounting**, **Updating**, and **Unmounting**.

Mounting:
Mounting means putting elements into the DOM.
React has four built-in methods that gets called, in this order, when mounting a component:

1. constructor()
2. getDerivedStateFromProps()
3. render()
4. componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

Constructor –

The constructor() method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

The constructor() method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

Example:

The constructor method is called, by React, every time you make a component:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}
```

```
ReactDOM.render(<Header />, document.getElementById('root'));
```

getDerivedStateFromProps –

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.

This is the natural place to set the state object based on the initial props.

It takes state as an argument, and returns an object with changes to the state.

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

Example:

The getDerivedStateFromProps method is called right before the render method:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}
ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));
```

Render –

The render() method is required, and is the method that actually outputs the HTML to the DOM.

Example:

A simple component with a simple render() method:

```
class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

componentDidMount –

The componentDidMount() method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

Example:

At first my favorite color is red, but give me a second, and it is yellow instead:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
```

```
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

Updating:

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's state or props.

React has five built-in methods that gets called, in this order, when a component is updated:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

The render() method is required and will always be called, the others are optional and will be called if you define them.

1. getDerivedStateFromProps

Also at *updates* the getDerivedStateFromProps method is called. This is the first method that is called when a component gets updated.

This is still the natural place to set the state object based on the initial props.

The example below has a button that changes the favorite color to blue, but since

the getDerivedStateFromProps() method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

Example:

If the component gets updated, the getDerivedStateFromProps() method is called:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));
```

2. shouldComponentUpdate

In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is true.

The example below shows what happens when the shouldComponentUpdate() method returns false:

Example:

Stop the component from rendering at any update:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return false;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

Example:

Same example as above, but this time the shouldComponentUpdate() method returns true instead:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return true;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

3. render

The render() method is of course called when a component gets *updated*, it has to re-render the HTML to the DOM, with the new changes.

The example below has a button that changes the favorite color to blue:

Example:

Click the button to make a change in the component's state:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}


ReactDOM.render(<Header />, document.getElementById('root'));
```

4. getSnapshotBeforeUpdate

In the getSnapshotBeforeUpdate() method you have access to the props and state *before* the update, meaning that even after the update, you can check what the values were *before* the update.

If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and after one second, the favorite color becomes "yellow".

This action triggers the *update* phase, and since this component has a getSnapshotBeforeUpdate() method, this method is executed, and writes a message to the empty DIV1 element.

Then the componentDidUpdate() method is executed and writes a message in the empty DIV2 element:

Example:

Use the getSnapshotBeforeUpdate() method to find out what the state object looked like before the update:

```
class Header extends React.Component {
 constructor(props) {
  super(props);
  this.state = {favoritecolor: "red"};
 }
 componentDidMount() {
  setTimeout(() => {
   this.setState({favoritecolor: "yellow"})
  }, 1000)
 }
 getSnapshotBeforeUpdate(prevProps, prevState) {
  document.getElementById("div1").innerHTML =
  "Before the update, the favorite was " + prevState.favoritecolor;
 }
 componentDidUpdate() {
  document.getElementById("div2").innerHTML =
  "The updated favorite is " + this.state.favoritecolor;
 }
 render() {
  return (
   <div>
    <h1>My Favorite Color is {this.state.favoritecolor}</h1>
```

```
      <div id="div1"></div>
      <div id="div2"></div>
     </div>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

5. componentDidUpdate

The componentDidUpdate method is called after the component is updated in the DOM.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and the color becomes "yellow".

This action triggers the *update* phase, and since this component has a componentDidUpdate method, this method is executed and writes a message in the empty DIV element:

Example:

The componentDidUpdate method is called after the update has been rendered in the DOM:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
```

```
 }
 componentDidUpdate() {
  document.getElementById("mydiv").innerHTML =
  "The updated favorite is " + this.state.favoritecolor;
 }
 render() {
  return (
   <div>
   <h1>My Favorite Color is {this.state.favoritecolor}</h1>
   <div id="mydiv"></div>
   </div>
  );
 }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

Unmounting:

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

- componentWillUnmount()

componentWillUnmount –

The componentWillUnmount method is called when the component is about to be removed from the DOM.

Example:

Click the button to delete the header:

```
class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = {show: true};
  }
  delHeader = () => {
    this.setState({show: false});
  }
  render() {
    let myheader;
    if (this.state.show) {
      myheader = <Child />;
    };
    return (
      <div>
      {myheader}
      <button type="button" onClick={this.delHeader}>Delete Header</button>
      </div>
    );
  }
}

class Child extends React.Component {
  componentWillUnmount() {
    alert("The component named Header is about to be unmounted.");
  }
  render() {
    return (
      <h1>Hello World!</h1>
    );
  }
}

ReactDOM.render(<Container />, document.getElementById('root'));
```