Microsoft: DEV211.1x

Introduction to HTML and JavaScript

Module 1: Creating Web Pages using Hypertext Markup Language

## Introduction to HTML

* intro to semantic markup
* using semantic markup
* Why HTML?
* HTML & XML
* Web Browsers
* HTML versions

## Document Structure & layout

* HTML Element
* BODY/HEAD Elements

## Basic HTML Elements

* text elements
* block elements
* inline elements
* images

## Formatting HTML Elements

* Semantic Formatting Elements
* Attribute Formatting

## Hyperlinks

* External Hyperlinks
* Internal Hyperlinks

## Intro to HTML

**Semantic Markup -** The process of annotating text so others can easily understand its meaning.

Example: take 3 words from the phonetic alphabet (Alpha, Bravo, Charlie)

typing them on one line doesn't get the idea across that it's a list. If we restructure the 3 words as follows:

# [list]

# [item]Alpha[/item]

# [item]Bravo[/item]

# [item]Charlie[/item]

# [/list]

It's much more clear that the text is a list of items.

Semantic Markup is still widely used today. In academia, it's common to use markup languages while writing dissertations, etc. While using a normal word processor, managing the appearance of your document can take up large amounts of time. Using a markup language allows the writer to focus on the content, rather than the appearance.

Here's an example that shows how to structure a table using Markdown markup with only text:

# age | name | grade

# --- | --- | ---

# 8 | Peter | 3

# 7 | Susan | 2

# 8 | Jessica | 3

If we run this markup through a parser, we'll get a table rendered however we choose, which means we can focus on the actual table data while writing it, and the colors/format can be revised after the document is complete. This vastly improves efficiency.

HTML is a markup language. It is rendered client-sided by the browser. We have now demonstrated that we can represent the meaning of our data by defining a list. We can modify the appearance of the list using CSS.

In 1989, Tim Berners-Lee worked at a lab named CERN. Scientists had already been using a markup language called Standard Generalized Markup Language (SGML). This language included annotations that allowed researchers to apply formatting to their documents quickly such as:

# <P>: paragraphs

# <H1>-<H6>: headers

# <TITLE>: titles

# <OL>: ordered lists

# <UL>: unordered lists

SGML's specification was very open, allowing different labs to use the language in different ways. This opened the door for multiple interpretations, causing incompatibilities across labs.

It also lacked the ability for documents to reference each other. Berners-Lee changed this when he created the HTML language - which is an extension of SGML that includes more annotations, and the ability to **link** one document to another. He went on to refer to the text and this design as hypertext and hypermedia respectively.

HTML is an extension of SGML, but it does some things differently:

* HTML restricts which SGML tags can be used on a web page
* HTML creates a dedicated tag (<a>) known as the anchor tag to reference other documents

**Links** within HTML popularized the language very quickly. Researchers could now find relevant information quicker than ever before by following these links; likewise, the ones composing these documents could link their work to others, creating a 'web' of information. In 1990, Tim Berners-Lee published a formal proposal titled WorldWideWeb that described a system of hypertext documents viewed by software named browsers in a client-server architecture.

Shortly after the proposal of HTML 1.0 & 2.0, Tim Berners-Lee founded and led the World Wide Web Consortium (W3C), whose purpose is to develop new standards for the World Wide Web on a regular basis. W3C has created specifications for HTML, XML, CSS and even XHTML. They've also developed other specs including: SOAP, SPARQL and MathML

### XML

Extensible Markup Language (XML), a W3C creation, was designed to be readable by both humans & machines. Much like HTML, XML is an extension of SGML; but further restricts its use by requiring all tags to be closed and attributes to be explicit. XML uses a strict markup language, intended to display data in a hierarchical format:

# <?xml version="1.0" encoding="UTF-8"?>

# <list>

# <item price="0.75">Apple</item>

# <item price="0.25">Banana</item>

# <item price="1.00">

# Orange

# <plucodes>

# <plucode description="Navel Orange">4012</plucode>

# <plucode description="Blood Orange">4381</plucode>

# <plucode description="unknown" />

# </plucodes>

# </item>

# </list>

This example shows a list of fruit at a store. There are different components, or **elements,** to this code. Each set of opening/closing tags (or empty tag) is considered an element. Elements can be nested within one another. In the code above, there are 3 item elements. The oranges element contains the <plucodes> element nested inside. This is known as a **child element**. The child element <plucodes> also has 3 <plucode> child elements. Each of these elements specifies a description as an **attribute** and the code as the content of the element.

### Constructs

Much of the terminology used to describe XML constructs are the same of that used in HTML. These will be the constructs this course covers.

A **tag** is an XML construct that begins with < and ends with >. below are the 3 tag constructs used in XML:

* + **Opening (or Start) Tag:**<item> These tags represent the beginning of a logical component in XML.
  + **Closing (or End) Tag:** </item> These tags represent the end of a logical component in XML.
  + **Empty Element Tag:** <item /> These tags represent both a logical component without any child content. (these tags stand alone. EX: <br /> for a line break) Empty element tags do not need to be closed in HTML5; however, in XML, all elements must be closed to be read by a parser. Best practice is to always close elements. The / always comes after the tag name and a space.

Let’s review some of the terminology we’ve learned by breaking down the fruit example code from earlier. The portions of code in bold below each term represent which part of the code the corresponding term applies to.

**Element:**

# **<item** price="0.75"**>**Apple**</item>**

**Child Element:**

# <item price="1.00">

# Orange

# **<plucodes>**

# **<plucode** description="Navel Orange"**>**4012**</plucode>**

# **<plucode** description="Blood Orange"**>**4381**</plucode>**

# **<plucode** description="unknown" **/>**

# **</plucodes>**

*Note: <plucodes> is a child element of the Oranges element, and each <plucode> element is a child of <plucodes>*

**Attribute:**

# <item **price="0.75">**Apple</item>

**Content:**

# <item price="0.75">**Apple**</item>

And finally, the **Declaration**:

# <?xml version="1.0" encoding="UTF-8"?>

This is the first line of code in an XML document. It specifies info about the doc including version and encoding.

**Remember –** HTML & XML are both derived from SGML. They restrict some language features, and add some others. Years later, a new HTML standard (XHTML 1.0) was created to apply the strict rules of XML to HTML. XHTML was the standard for years, but was eventually replaced by the current standard, HTML5.

### Browsers

Web browsers are the gateway to web apps. They perform 3 primary functions:

* Retrieve contents over HTML apps when given a Universal Resource Locator (**URL**)
* Structures the web page in an aesthetically pleasing format, as opposed to raw HTML
* Allows us to follow hypertext links within a page or externally, just by clicking

There are many different browsers available. Edge now ships with Windows (peace IE), Safari with Apple as well as a plethora of other browsers that can be downloaded on either OS, including: Firefox, Google Chrome, Brave and Vivaldi. They all can display the same web content, but serve as a different way to view the content. With so many different browsers & operating systems to choose from, developers tend to use HTML markup, CSS stylesheets & JavaScript as a cross-platform medium to eliminate most OS/app/device specific compatibility issues. We’ll focus on writing open & accessible HTML code for that purpose.

### History of HTML versions

**SGML**: Paved the way for HTML. Was used to structure text so writers could focus on the content of their work rather than its style. Used paired tags. See table on page 2.

**HTML 1.0**: Developed in 1991. Used some paired tags from SGML, biggest feature is the introduction of hypertext links, allowing documents to reference one another. Each anchor (<A>) element contained an HREF attribute specifying the location of another document. This is when the specification for the familiar [www.site.name](http://www.site.name) format was invented to address remote machines. Soon after, early browsers like Mosaic were created to explore the web.

**HTML 2.0**: Developed in 1995. Interest in HTML spiked after release of HTML 1, and with it came many ideas/revisions to the standard. The Internet Engineering Task Force assembled a group to develop a new HTML spec, which became HTML 2.0. Was later updated with features like tables, image maps,and internationalization and file uploads. Just before its release, Internet Explorer and Netscape made their debut.

**HTML 3.0-3.2**: Developed in 1997 by the newly formed W3 Consortium. Browser vendors begin to get involved in development process. New features include: frames, applets, wrapping text around images (flow), and additional text elements. JS & CSS styling begin to take shape.

**HTML 4.0-4.0.1**: Developed in 1999, this version removed lots of browser-specific formatting markup to promote the cross-platform & browser uniformity through CSS. Also, enabled control over the amount of deprecated elements (such as marquees and frames) allowed on your web page.

**XHTML 1.0**: Developed in 2000. Basically, just rigorously enforced a variety of the strict rules of XML to HTML. Another move towards uniformity. Many XHTML pages still exist, but the standard is no longer supported.

**HTML5**: Developed in 2014. Latest version of HTML, and the one used in this course. Added many new semantic elements and attributes. Took a lot longer to develop as it removed lots of SGML features to create a spec with a clear path forward, hopefully lasting much longer than its predecessors.

## Document Structure & Layout

### HTML Element

The HTML element is what tells the computer that the document is written in HTML.

Check out this code:

# <!DOCTYPE html>

# <html xmlns="http://www.w3.org/1999/xhtml" manifest="thisapplication.appcache" lang="en-US">

# </html>

Let’s break it down and examine what each section means.

# <!DOCTYPE HTML>

Above is the first line of code in an HTML doc. Doctype declares the doc as HTML, and is required before the HTML tag.

# <html

HTML tags are the root element of the doc. All other tags are nested inside of it.

# xmlns=http://www.w3.org/1999/xhtml

This is called the xmlns attribute, and it specifies the namespace(?) for HTML file. Prev, version of HTML(and XHTML) required this, but in HTML5 it’s optional. Will default to *http://www.w3.org/1999/xhtml* if not included.

# manifest="thisapplication.appcache"

This attribute specifies location of cache manifest. Allows apps to cache info offline. Pages load faster, and can be viewed offline.

# lang="en-US">

global attribute, specifies the language of the element it is included in. Typically, this is paired with the root html tag to specify the language for whole doc.

# </html>

Closing tag – all HTML elements in the entire document are inside of the HTML tag.

### BODY/HEAD Elements

# <head>

# <title>Sample Page</title>

# </head>

# <body onload="handleLoad" ononline="handleNetwork" onresize="handleResize">

# </body>

Code breakdown:

# <head>

Container for **metadata**, which is a set of data that describes other data. Placed between the <html> and <body> tag.

# <title>Sample Page</title>

The content inside of the title tag represents the title of the HTML page.

# <body

This tag represents the root of the actual content rendered on the HTML page.

# onload="handleLoad"

This attribute specifies a JavaScript method to be called when the page finishes initial loading.

# ononline="handleNetwork"

attribute that specifies another JS method, is called when network connection is available for page.

# onresize="handleResize">

attribute that specifies JS method that’s called when page is resized. Mostly when resizing the browser window.

## Basic HTML Elements

### Text Elements

The “Lorem ipsum..” sentence is intentionally nonsensical, the purpose is to examine the design or layout of the text without being distracted by trying to read and interpret the text. The sentence is Latin, and has been used by typographers since the 1500s.

Some text elements used in <body>:

**<p>** - paragraph element, used to mark where a paragraph begins. To begin another paragraph, must close </p> and start another <p> otherwise, they will not be separated.

**<b> -** bold

**<i> -** italics

**<ins>** - insert. Underlines text. Used to show where text was added

**<del>** - delete. This is a strikethrough. Used to show what text was removed.

**<mark>** - highlights text

**<br />** - line break. Note that it is an empty element.

**<h1>** - headers – this is the 1st one

**<h2>**, **<h3>**, **<h4>**, **<h5>**, **<h6>** - other headers. The higher the number, smaller the header.

**<strong>** - bold – implies semantic relevance for text that is *strong*

**<em>** - italics – implies semantic relevance for text that deserves *emphasis*

**<small>** - shrinks text to a smaller font

**<sup>** - superscript; text is smaller and appears higher on the line – EX: 1st ‘st’

**<sub>** - subscript; text is smaller and appears lower on the line – EX: H2O

### Block Elements

There are 2 element container types in HTML: Block Elements & Inline Elements. The type dictates how its content will be rendered. Block-level elements take up the full width of the screen. These are always rendered on a new line. The <br>, <body> and <p> elements are all block-level elements since each one takes up the entire width of the screen each time they are used.

The <div> element is a multi-purpose element. Basically a container for other content. Can nest many div elements within each other. Create siblings with content rendered on different lines. A few block elements are: <article>, <aside>, <blockquote>, <br>, <buttons>, <canvas>, <caption>, <dd>, <dl>, <dt>, <h1>, <h2>, <h3>, <h4>, <h5>, <h6>, <footer>, <header>, <fieldset>, and <form>

Elements like <article>, <header>, and<footer> are more universal than most <div> elements. As such, these should be used over others whenever possible so other developers reading your work can easily infer the meaning of your text.

### Inline Elements

Inline elements, unlike block elements, only take up the amount of width on the page that they need to render their content. Multiple inline elements can render on the same line, providing their combined length is shorter than the width of the line they’re on. If they’re too long to fit on one line, they will wrap their content to the next line.

The most common inline element is <span>, but there are many others including: <strong>, <ins>, <del>, <sup>, <sub>, <i>, and <em>

Inline and Block elements can be used together as siblings, but typically you can only place inline elements within block elements, as block elements always take up the entire length of the line and inline elements only take up what is needed. There is a CSS attribute called **display** that allows you to change the rendering of any element between block and inline, however it’s not recommended to modify the way HTML elements are expected to behave.

### Images

Images are represented using the <img> tag in HTML. This tag provides metadata about the image file and preferred sizing to assist the browser in rendering the image. Here’s an example img tag:

# <img src="C:/Users/kaele/Pictures/dispreport.png" />

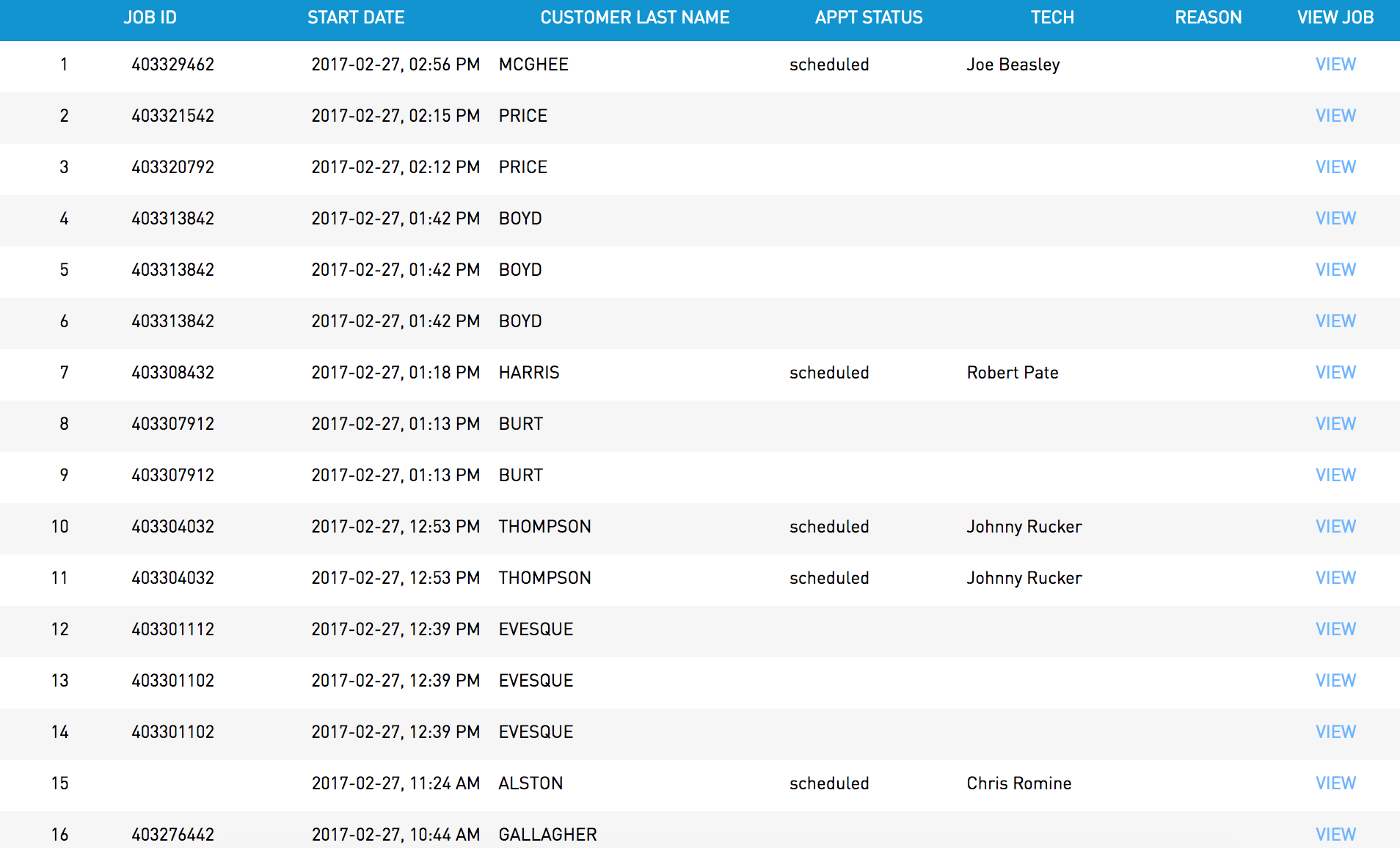
Calling an image like this will use the images original size, but we can modify the size & aspect ratio of the image using the height and width attributes. Adjusting only one of these attributes will change the image size while maintaining its aspect ratio. For example, if an images is 130x129 pixels, and we set a width attribute of 350:

# <img src="C:/Users/kaele/Pictures/dispreport.png" width="350" />

The image will adjust its height to 347.30 to maintain the original shape.

Whereas, specifying the width *and* height attributes will resize the image without locking the aspect ratio, often resulting in warped images:

# <img src="C:/Users/kaele/Pictures/dispreport.png" width="350" height="120" />



You can set alternate text for an image using the alt attribute:

# <img src="C:/Users/kaele/Pictures/dispreport.png" alt="random pic" />

This text is only rendered if there is a problem loading the image. For example, if the image src points to an empty location.

## Formatting HTML Elements

In HTML, we can write text by simple just writing it. When it’s rendered, it will show up as standard text in the default font of the browser. If we want to change the appearance of the text, we’ll have to add formatting elements such as <b>, <i> or <mark>. These elements can be used together to alter the appearance of text in the browser.

### Semantic Formatting Elements

**Semantic Formatting Elements** are additional formatting elements that appear to offer the same functionality as other elements (such as <strong> and <b>), and they do render the same in the browser. The only difference between these two is the semantic meaning. Where <b> represents a block of text rendered in bold type-font with no additional meaning, the <strong> tag indicates that the text is semantically ‘strong’.

This is also true for the relationship between the <i> tags and the <em> tags, where <em> stands for *emphasis*. <strong> and <em> can have their styling reassigned through CSS. For example, we could change <strong> to render text as 3 font sizes larges, or use purple text.

### Attribute Formatting

Ideally, HTML dictates semantic structure of a web page, while CSS control its look and feel. Remember, we want to keep these two aspects separate so we can maintain focus on each task individually.

Prior to HTML5, it was common to see tags denoting style for the document. This overcomplicated document styling, as you had to go in and edit the HTML code to change the appearance; where using CSS allows you to switch the sheet and change the entire look & feel of the document.

Here’s an example of how someone may have structured an HTML document inline before HMTL5:

# <!DOCTYPE html>

# <html lang=”en-us”>

# <head>

# <title>HTML Page</title>

# </head>

# <body **bgcolor=”lightblue”**>

# <p>

# <**font face=”Consolas” size=”62px” color=”purple”**>

# Text

# </**font**>

# </p>

# </body>

# </html>

The text in bold represents the antiquated document styling method. Imagine you have a 25 page document where text is styled using <font face=”Consolas” size=”62px” color=”purple”> over 50 times. If you wanted to change the font size for each occurrence of this code, you’re going to spend a lot of time manually locating and typing out the desired size, where using CSS, you could assign this styling to the element <strong>, and change the font one time – automatically applying this change to each instance of <strong>.

It’s important to understand these outdated methods of styling in HTML so we know not to incorporate them into our code. If we see these being used in any modern code, we should change them to reflect today’s HTML/CSS standards & educate the writer on the new standards.

## Hyperlinks

Hyperlinks will direct you to either another page or a specific spot on the given webpage. They allow you to navigate between websites by interacting with clickable words, phrases or images. In this section, we’ll learn how to create external and internal hyperlinks.

### External Hyperlinks

Hyperlinks connect different web pages, creating a ‘web’ of documents; this is why we know the internet as the ‘World Wide Web’.

To create a hyperlink in HTML, we use the <a> tag, but now we must assign attributes. There are many attributes that can be assigned to the <a> element. Let’s start with *href*:

# <a href=http://www.website.com”></a>

This attribute specifies the website that the link will lead to.

Another attribute is the *target* attribute:

# <a href=http://www.website.com” target=”\_self”></a>

This attribute specifies the browser window behavior when the user clicks the link. Will the link open in a new tab, replace the current tab or even open in a new window? \_self is assigned as the target by default if not specified, which replaces the current tab. Another specification for the target attribute is *\_blank* which opens the link in a new tab.

Now let’s look at *rel*:

# <a href=http://www.website.com” target=”\_self” rel=”author”></a>

This is used to specify the relationship between the document that the link is on and the one that the link leads to, such as a link to an author or to a help page.

The *download* attribute:

# <a href=http://www.website.com” target=”\_self” rel=”author” download></a>

Use this attribute when you want to specify that the user download the contents of the page the link points to, rather than viewing it in a browser. Can be used without specification

Now we’ll add some text to this hyperlink. This how we create the clickable text that the end user sees:

# <a href=http://www.website.com” target=”\_self” rel=”author” download=”download”>**This is the link text**</a>

### Internal Hyperlinks

Internal hyperlinks are used to reference a specific section in long documents. For example, a Wikipedia page uses internal hyperlinks in its contents section at the top of the document. We can create internal hyperlinks by using the *id* attribute:

# <p id=”writeWhateverHere”>This is a paragraph that will be referenced with a link.</p>

Now that we’ve established our ID we’ll need to reference it through the anchor element:

# <a href=”#writeWhateverHere”>This is a link that leads to the writeWhateverHere ID.</a>

You can also link directly to a specific ID point on the document from an external page:

# <a href=http://www.website.com/index.html#writeWhateverHere></a>

## Module 2: Creating Sophisticated Document Structures using HTML

## Capturing User Input

* Input Controls
* Form Element
* Submit Button
* HTML5 Field Validation
* Fieldsets and Legends
* HTML5 Form Elements

### Input Controls & Submit Button

**<textarea>** element – This element renders multi-line text input control, allowing the user to input lots of text that can wrap & span across many lines:

# <textarea rows="3" cols="30">

# We live in a mobile-first and cloud-first world. Computing is ubiquitous, and experiences span devices and exhibit ambient intelligence.

# </textarea>

It’s essentially just a sizable text-input box from what I can see

The **<select>** element is for drop-down boxes with multiple options. The **<option>** element is the child element used in conjunction with <select>:

# <select>

# <option value="DAT202x">Processing Big Data with Azure HDInsight</option>

# <option value="DAT204x">Introduction to R Programming</option>

# <option value="CLD203x">Office 365: Managing Identities and Services</option>

# <option value="DEV208x">Introduction to jQuery</option>

# </select>

The content of each option attribute is what will display as the drop-down options to the end user, but notice also there is a **value** attribute. This is what will be saved to the server if the user selects that option.

The **<optgroup>** element allows you to group similar options together. This is useful when dropdown lists have many options:

# <select>

# <optgroup label="Data Platform">

# <option value="DAT202x">Processing Big Data with Azure HDInsight</option>

# <option value="DAT204x">Introduction to R Programming</option>

# </optgroup>

# <optgroup label="Cloud">

# <option value="CLD203x">Office 365: Managing Identities and Services</option>

# </optgroup>

# <optgroup label="Development">

# <option value="DEV208x">Introduction to jQuery</option>

# </optgroup>

# </select>

The **label** attribute specifies the group name.

The **<input>** element is the most basic amongst HTML form elements and can be used to capture a variety of user input, including text and numeric values. It has a **type** attribute, which allows you to specify metadata about the field:

# <form ...>

# Username: <input type="text" name="username" />

# E-mail Address: <input type="email" name="emailaddress" />

# </form>

In the example above, we also used the **name** attribute, which is used to reference elements in JavaScript or form data after a form is submitted. They’re basically variable names. Input is an empty element.

The **value** attribute can be used to specify a field’s default value. This will pre-fill a text field with whatever it is specified as. This is not to be confused with the **placeholder** attribute, which also puts text into the field, but when the user clicks in the field, the text disappears. This is useful to specify to the user what is supposed to be typed into the field.

There are a wide variety of types that can be used with the input element. Here are a few:

**Button** – this will render a button. Usually used to execute JavaScript code.

**Checkbox** – This renders a checkbox which can be independently selected. It can only be either on or off.

**File** – This renders a ‘Choose File’ button which prompts the user to select a file from the client device.

**Hidden** – hidden does exactly what it sounds like, it hides the input field from the user. This is used to store default values, such as a country field whose users are all from the same country. It is redundant for the user to type their country, but the data may need to be recorded for another reason. Hidden can have its value changed by a JavaScript

**Image** – use this to render an image that you specify as the ‘submit’ button

**Password** – This will render a text input that masks characters for privacy

**Radio** – this renders a radio button that can be selected amongst a group of radio buttons

**Text** – this renders a basic text field.

**Submit** – when clicked, this renders a button that submits the entire HTML form.

Let’s move on to the **<button>** element. This element is similar to the <input> element’s attribute type button, but using the <button> element you can put content like text or images within the button. Always specify type for <button>:

# <button type="button">

# <img src="images/freeAgent.png" alt="Entry Level Programmer" />

# <br />

# Hire Me!

# </button>

### Form Element

We use forms to capture user input and submit it to different servers to process for later use. We can do many different things with this element. We can specify where the form data is sent using the **action** attribute:

# <form action=”http://www.website.com/formDataStore.html”>

The action specified above is known as an *absolute URL,* it can be used from any website

If our form was located on the same website as the page where the form data will be stored, we could use what’s known as a *relative URL*, which simply uses the file name:

# <form action=”formDataStore.html”>

Now that we’ve got our storage location set, we must specify *how* it will be sent there with the **method** attribute. There are two values we can set for this attribute – **get** and **post**. The get value will append the form data into a URL by pairing each name attribute specified (remember, basically variables) with the value the user entered. For example:

# <form action=”http://www.website.com/formDataStore.html” method=”get”>

If we used the attributes above with the username/email code from earlier, it might yield the following url:

*https://www.website.com/formDataStore.html?username=glorginax&emailaddress=poopy47@lame.net*

The downside to using the get value are that, obviously, form data is visible in the URL. This is a no-no for forms that might contain private information. Also, URL length is limited to 3000 characters.

Using post for your method attribute’s value, on the other hand is more secure as form data is appended inside the body of the HTTP request. Post does not have any character limitations, but form submission cannot be bookmarked

**<fieldset>** is a form element that can be used to pair sets of fields in forms:

# <form ...>

# <fieldset>

# <legend>Names</legend>

# <input type="text" name="firstName" placeholder="First Name" />

# <br />

# <input type="text" name="lastName" placeholder="Last Name" />

# <br />

# </fieldset>

This will put a square around these fields, setting them apart from others. **<legend>** is a child element of <fieldset> and is used to label each fieldset area.

### HTML5 Field Validation

This was introduced into HTML5 as a means of validating that all required information had been entered prior to submission. If validation requirements are not met, the form is not saved. There are a number of validation attributes for the <input> element:

**Required** attribute requires that a text input has valid data before the form is saved:

# <input type="text" name="firstName" required=”required” />

**Pattern** attribute specifies an expression that the value of the field must match before form can be saved:

# <input name="zip\_code" type="text" pattern="\d{5}(-\d{4})?" required="required" />

**Readonly** attribute specifies that a field is read only, and can’t be edited by the user:

# <input name="profile\_url" type="url" readonly="readonly" />

**Disabled** attribute specifies that a field is disabled and cannot be modified by the user:

# <input type="submit" disabled="disabled" />

**Min** and **max** attributes are used the **range** input type, and are used to specify the boundaries for the selected numeric value:

# <input name="issue\_quantity" type="range" min="1" max="15" />

They ca also be used with **date** inputs:

# <input name="service\_date" type="date" min="2000-01-01" max="2999-12-31">

**Autocomplete** attribute is used with various text inputs to toggle autocomplete feature. Works with most browsers:

# <input name="api\_key" type="text" autocomplete="off" />

New input types in HTML5:

**Color** – This renders a color picker.

**Date** – This renders a date control that allows you to pick a year, month and day without a specific time.

**Datetime-local** - This renders a date control that allows you to pick a year, month, day and time without time-zone info stored.

**Email** – validates that the email address is valid.

**Month** – renders a date control that allows you to select year and month only.

**Number** – renders a text input field that only allows numeric input.

**Range** – this renders a slider control that allows users to select an imprecise number.

**Search** – This renders a text input field used for search.

**Tel** – this renders a text input field for telephone numbers.

**Time** – this renders a control that allows you to enter time without saving time-zone info.

**url** – renders a text input field that validates whether the URL is valid or not.

**Week** – this renders a date control that allows you to select year and week only.

You can always skip validation if you have your own validation code by using:

# <form novalidate="novalidate">

# </form>

## Capturing User Input

We put most of our HTML doc content in <p> or <div> tags. It’s convenient, but they don’t have any obvious meaning. These are known as **non-semantic elements**. In other words, someone could not infer its meaning by simply looking at it. As a workaround, many people would provide large amount of comments, and very descriptive CSS class names. This created issues – one being that it can greatly increase the size of your webpage, which is obviously bad.

### Common Semantic Elements

HTML5 combated the issue of non-semantic elements by introducing new semantic elements with the same functionality as other block-level elements & should always be used over their non-semantic counterparts: **<header>, <footer>, <nav>, <section>, <article>, <aside>, <details>, <figure>, <figcaption>, <main>, <mark>, <summary>, <time>**

These elements can be used anywhere you’d use a block-level element.

Each of these new semantic elements has its advantages. Like the **article** element; a search engine will search through the content of an article tag, but may ignore content in a header or nav tag. This increases search engine efficiency. Also, it has advantages in CSS. Not quite sure what any of this shit means yet, so I’m going to copy and paste it there then put it in my own words when we actually get to the CSS module:

“Another advantage of semantic elements is the ability to use CSS to design the element directly without requiring additional CSS classes. You can use features such as *flexbox* in CSS to put an **aside** element on the right-hand side of your webpage. You can also use features such as *grid* in CSS to place the **nav** element on a row by itself. More importantly, you can use features such as CSS Media Queries to re-arrange your semantic elements for a mobile browser.”

### Time Element

**Microformats** are elements in HTML that specify additional metadata without changing the format of the HTML document. We can add styling to these elements with CSS, we can handle them with some JavaScript or we can do neither of those & just leave them as semantic inferences for machines or other developers.

For example, say we just visited the doctor, and we want to make our next appointment date & time to be semantically inferable by a machine. Simply typing out the date and time as content in a paragraph element may be easily inferred by a human, but not a machine. To accomplish this, we’ll need to parse the information to a machine-readable format. The **<time>** element is a perfect microformat for this:

# <time datetime=”2017-08-04T08:00-08:30”></time>

In this format, the machine now knows the exact date & time of day for the next appointment.

### Figure Element

The **<figure>** element renders a self-contained image on the HTML document. Its child element, **<figcaption>** is used to provide a text description of the image that appears on the document.

## Scalable Vector Graphics

SVG is an image format with the ability to draw 2d graphics directly into the browser. It’s based on XML, and it supports features like interactivity, transitions and animations. SVG’s are normally stored in XML files and are edited with vector-based image editors. Browsers display SVG images similar to the way they display other image formats. In HTML5, we can now embed SVG images directly in the web page. We can create SVG shaped on the page, and manipulate them with CSS & JavaScript.

### SVG Shapes

HTML5 includes some basic SVG shapes we can embed into a web page without having to learn drawing paths. We can combine various shapes to create complex objects:

# <svg height="200" width="400">

# <rect x="100" y="50" rx="20" ry="20" width="250" height="100" fill="#1B043C" />

# <rect x="100" y="50" width="200" height="100" fill="#1B043C" />

# <circle cx="100" cy="100" r="50" fill="#472772" />

# <text fill="#D7BDD3" font-size="28" font-family="Segoe UI Light" x="160" y="108">MCS</text>

# </svg>

This combination of SVG shapes renders an image that looks like a company logo.

The **<circle>** element creates a circle with a defined center point & radius. Here are some attributes we can use with the circle element & a short description:

**cx & cy** – These attributes are used as a pair of coordinates to define the location on the screen where the center of the circle will render. Default is (0, 0)

**r** – This attribute is used to define the radius of the circle.

**fill** – quite obviously, this attribute defines the color for the circle’s interior.

**stroke** – This defines color used for the border of the circle

**stroke-width** – this is used to define the thickness of the border around the circle.

The **<rect>** element renders 4-sided rectangular polygon that is drawn from a defined from a top-left point, width and height. It has the same attributes with the following exceptions:

**x** & **y** - these <rect> attributes are counterparts to that of the circle element’s cx & cy attributes. They define the coordinates for the top-left of the rectangle. Default is also (0, 0)

**rx** & **ry** - attributes are used to round the edges of the polygon on the x or y axis.

**width** – this defines the rectangle’s width

**height** – this defines the rectangle’s height

**fill, stroke** & **stroke-width** are all used the same way as they are used for the circle element.

The **<polygon>** element is used in conjunction with the **points** attribute. This element allows you to specify 4 sets of coordinates - one for each corner – to create a customer shape. You can also use the following attributes: **stroke, stroke-width** & **fill**.

**<text>** is an SVG element that renders text from a defined bottom-left point using the **x** & **y** attributes. The **fill** attribute is used with this element to define the color of the text. This element also utilizes two font attributes:

**font-size** – this defines the size of the font

**font-family** – this defines the font of the text.

### SVG Element

The SVG element is the root of all SVG shapes that will render on the page. All SVG shapes must be located inside of this element. You can define **width** and **height** attributes for the SVG element, which specifies the dimensions that child elements can take up.

## Media Elements

HTML5 supports multimedia playback & comes with a new set of cross-browser supported media elements, which allow you to specify video/audio sources – along with parameters to play video or audio directly on the HTML page. You can specify multiple sources for media files so the browser can pick the file it natively supports. You can even pick which aspects of the media player are available via using attributes like **controls**.

The **<video>** element is used to play video files:

# <video height="500" controls="controls" poster="screenshot.png">

# <source src="advertisement.webm" type="audio/webm" />

# <source src="advertisement.ogg" type="audio/ogg" />

# <source src="advertisement.mp4" type="audio/mp4" />

# </video>

The **poster** attribute seen above specifies the location of an image file which will be used as the cover image on the media player for the video prior to it being played.

The **<audio>** element is used to play audio files:

# <audio autoplay="autoplay" controls="controls" loop="loop">

# <source src="popopen.wav" type="audio/wav" />

# <source src="popopen.ogg" type="audio/ogg" />

# <source src="popopen.mp3" type="audio/mpeg" />

# </audio>

Listing multiple sources is always the best route, as the browser will try each source in order from top to bottom until it finds a compatible source file.

If the browser does not support any of the file types listed in your sources, the audio/video elements content will be displayed. This is intended to be used as an error message in instances where the browser loops through all listed sources & still cannot find a supported file type:

# <audio autoplay="autoplay" controls="controls" loop="loop">

# <source src="popopen.wav" type="audio/wav" />

# <source src="popopen.ogg" type="audio/ogg" />

# <source src="popopen.mp3" type="audio/mpeg" />

# **This browser does not support the audio format.**

# </audio>

### Video/Audio Formats

There are 5 primary video/audio formats used in major browsers. The tables below show which browsers are compatible with these formats:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **HTML Video Formats** | | | | | | | |
| **Format** | **Media Type** | **Edge** | **Internet Explorer** | **Chrome** | **Firefox** | **Safari** | **Opera** |
| MP4 | video/mp4 | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| WebM | video/webm |  |  | ✔ | ✔ |  | ✔ |
| Ogg | video/ogg |  |  | ✔ | ✔ |  | ✔ |
| **HTML Audio Formats** | | | | | | | |
| **Format** | **Media Type** | **Edge** | **Internet Explorer** | **Chrome** | **Firefox** | **Safari** | **Opera** |
| MP3 | audio/mpeg | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| Ogg | audio/ogg |  |  | ✔ | ✔ |  | ✔ |
| Wav | audio/wav |  |  | ✔ | ✔ | ✔ | ✔ |

## Module 3: Design Web Page Elements using Cascading Style Sheets (CSS)

CSS is a language that is used to format the look and feel of an HTML document. With CSS, we can control things like font, font size, color, text alignment, background, margins and much more. We can manage pages much easier with CSS as it separates the HTML element from display information. Using CSS enables quicker downloading of web pages & provides a method for retaining a common style.

## Introduction to CSS

We can code CSS style rules in 3 places:

**inline**, or inside an HTML tag

An **Internal Style Sheet** is coded within the <head> element, and closed by the <style type=”text/css”></style> tags.

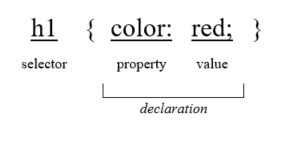
An **External Style Sheet** is a separate .css file used as a reference for multiple HTML pages. A link is defined in the HTML header that points the browser to the .css file.

### CSS Syntax

The two parts to a CSS style rule are:

**CSS selectors**: defines the HTML element being manipulated with CSS code.

**The declaration**: This consists of on or more **property-value** pairs. **Property** is the CSS element or style being manipulated, and **value** is the value of the specified property. Property-value pairs usually end in a semi-colon and the declaration (property *and* value) are enclosed in curly brackets:



## Style Definitions

This section will cover different style options in CSS, and their definitions.

### HTML Element Styles

Let’s break down some code and go over the following selector types:

# **body** {

# font-weight: bold;

# }

# **header**, **footer** {

# background-color: black; color: white;

# }

# **section > nav** {

# background-color: darkgray;

# }

# **section article** {

# background-color: lightgray;

# }

We’ll start with **body** – This will apply styles to each body element on the HTML page. There’s usually only one.

Next up: **header, footer** – This selector styles every header & footer element on the HTML page. By using commas, we can easily define multiple elements which our styling will apply to in our selector.

**section > nav** – is known as a **Direct Descendant Selector**, and is used by placing **>** between elements in the selector. This applies the styling only to direct descendant child elements (element after the **>**) of the first defined selector element (element before the **>**); in this example, the styling only applies to direct **nav** child elements of a **section** element.

**section article** – This is called an **Indirect Descendant Selector**, and it’s basically the same as using the **>** in the selector, but when it is omitted, the styling applies to (in this example) *all* <article> descendants of the section element, rather than explicitly to direct descendants. You can use multiple elements, placing a **>** in between each. This is used when you have an extremely specific instance of nested elements and only need styling applied to specific ones.

You can even combine the Direct and Indirect selector types:

# section > article p {

# font-style: italic;

# }

### CSS Class Styles

In CSS, we can create a class with its own style code:

# .purpleBackground {

# background-color: #B2A4A1;

# font-weight: bold;

# font-size: 2em;

# text-align: center;

# }

In the example above, we created a CSS class named ‘purpleBackground’ that will apply a purple background, bold text, change text size and align it to the center of the object. To apply this class to an HTML element, we must add the **class** attribute to an element:

# <article class=”purpleBackground”>

but say we want to use this class to make text large for headers, but we only want the other attributes any other time we call this class. There’s a way to accomplish this:

# .purpleBackground {

# background-color: #B2A4A1;

# }

# header.purpleBackground {

# font-weight: bold;

# font-size: 2em;

# text-align: center;

# }

by splitting this into two classes, we can add additional styling to **<header>** elements that specify the purpleBackground class attribute while any other element calling the purpleBackground class attribute will *only* get a purple background.

We can then even take it a step further and use our selectors when creating classes for very specific cases:

# header.purpleBackground > h2 {

# font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;

# }

This will apply a different font to an h2 child element of a header element whose class attribute is set to purpleBackground. If we were to set the purpleBackground class attribute on any element other than a <header>, this style will not apply, because it’s set to only apply used with header element.

### ID Style Selectors

A **CSS ID** is a unique identifier for an element, meaning it can only be used once in the document; unlike classes, which can be used multiple times within a document. A **CSS Style ID** is used to style the layout elements of an HTML page, and is usually preceded with a ‘#’. We can use CSS to identify and style any ID attribute however we like.

CSS style IDs are custom styles designed to work with div tags. All content inside the div tag follows your custom style. We can create these custom styles with declaration sets in the CSS document.

We can use IDs within an element in many creative ways, including writing JavaScript that can identify and manipulate the element or its content. The ID attribute can even replace anchor tags with name attributes, serving as a perfect target for internal links. Some people use the ID attribute as a documentation method, marking things like the beginning, middle and end of a large article.

IDs are used as fragment identifiers (an href attribute that ends in #anchor directs one to id="anchor"), and for JavaScript’s getElementById.

## Style Inheritance

**CSS style inheritance** is the process in which child elements inherit the styling of their parent element. Some properties are automatically inherited, while others - like margins or borders – cannot be inherited as they could conflict with the elements that are already set to the child elements.

Inheritance makes it easy to specify properties like font, or font size to apply to an element and all its children.

Inheritance is separate from the cascade, because inheritance only applies to the **Document Object Model** (DOM), which is the structure of an HTML or XML document. The cascade is only concerned with stylesheet rules. CSS properties attached to an element via cascade *can* be inherited by that element’s children, though.

All elements in an HTML doc inherit every property that they can, except for elements who do not have parents. Most characteristics can be automatically inherited in CSS, but what if you want the inherited property to carry more weight? This is possible through **Inherit Property Value**. We can increase the weight of the property inherited by adding it to the author style sheet.

Through inheritance, we can set characteristics to multiple elements without having to specify element property for every elements type, thereby making it possible to apply styles to a tag without directly applying to that particular HTML tag.

### Style Inheritance Order

CSS styles can come from lots of places:

**Browsers (User Agent)** – Browsers have default stylesheets that determine how the page and elements are rendered for the end user.

**User Options** – Users can change certain aspects of the browser, like font size for readability. The setting is usually saved as a stylesheet which overrides the browser’s default stylesheet.

**Author CSS** – CSS that is custom written by the author of the HTML page.

CSS styles in an HTML page are organized in an ascending order of inheritance. User options will override any default styling set by the browser (User Agent), and author CSS will override both User Agent *and* user options.

In CSS, we can use the ‘**!important**’ keyword to override existing CSS styles, effectively skipping to the front of the order of inheritance. For example, we could use the following code to ensure no other styles could affect our red text:

# body {

# color: red !important;

# }

End users can also specify styles using the !important keyword to override the author of the page. This is useful for people with colorblindness, or vision impairments who want enlarge text or change a page’s color.

Now that we threw !important into the mix, we’ve got 5 styles in the inheritance queue now, sorted as follows:

1. User !important declarations – highest priority
2. Author !important declarations
3. Author CSS
4. User settings
5. browser default stylesheet – lowest priority

### Combining Multiple CSS Styles

We can combine multiple CSS styles by naming both styles in the HTML element’s class attribute. Look at these styles:

# article.spacer {

# padding: 15px;

# margin: 25px;

# border: thin lightgray solid;

# }

# .babyBlueBackground {

# background-color: #57BCF4;

# }

We can apply the spacer class with the babyBlueBackground class:

# <article class="spacer babyBlueBackground">

# Lorem Ipsum

# </article>

Now the content of this article element will have both the babyBlueBackground and spacer styles applied to it; but remember, naming the CSS class article.spacer means that when we call that class in an HTML element, its CSS declarations will only apply to article elements. Therefore, if we called this class in a section element:

# <section class="spacer babyBlueBackground">

# Lorem Ipsum

# </section>

The spacer styling would not apply, but the babyBlueBackground styling would This is because the .babyBlueBackground class is not restricted to any specific element, it just applies itself any time it is called.

## Box Model

### Margin and Padding

The **margin** property in CSS defines spacing between elements in HTML. It’s a **shorthand property**, which means it sets the value of multiple properties simultaneously. There are four parts to the margin property: **margin-top**, **margin-right**, **margin-bottom** and **margin-left**. Each defines its respective margin in pixels (px). There are a few ways to define margins in CSS:

**CSS margin:** – this method can use 4 values, which are defined in order of: top, right, bottom, left. EX:

# margin: 10px 10px 10px 10px;

or, we can only define two of these values, which applies the first value to the top and bottom margins, then the second value to the left and right margins:

# margin: 10px 10px;

**CSS margin: value** – We can take the above method a step further by defining only one value:

# margin: 10px;

obviously, this will set a 10-pixel margin for all four sides.

**CSS margin: margin-**[**direction**] – This is how you’d set a value for one direction, and ignore the others:

# margin-left: 10px;

The margin property can also be set to negative values; but if not defined, margins will default at 0. Some browsers have default margin distance setting for certain elements (like <p>). We can remedy this by setting the margin to 0.

The **padding** property in CSS defines the amount of space that appears between an element’s content and its border. We can use this property to change the default padding value for various elements like *paragraph*or*table*. Padding rules are essentially the same as that of margin, except that you cannot declare negative or auto values for padding.

We’ve got the same four properties: **padding-top**, **padding-right**, **padding-bottom** and **padding-left**.

We can define padding using four values (same direction order as margin):

# padding: 10px 10px 10px 10px;

Two values (top & bottom):

# padding: 10px 10px;

one value (all directions):

# padding: 10px;

and padding-[direction] to select only one direction to set a value for:

# padding-left: 10px;

### Borders

Let’s go over some border declarations in CSS.

**border-width** – this property defines the thickness of the border itself, and its value is measured in pixels:

# border-width: 5px;

**border-color** – this property defines, you guessed it, the color of the border, and its value uses hex color code:

# border-color: #69D2E7;

**border-style** is a bit different. There are a few values we can set in this declaration to change the look of the border:

# border-style: solid;

other values we can use in this declaration include: **dotted**, **dashed**, **double**, **groove**, **ridge**, **inset** and **outset**. **Groove** & **ridge** are inverse styles, and so are **inset** & **outset**.

There’s also a **hidden** value for the **border-style** property, which renders any spacing or thickness for the border, but the border itself is not visible. Finally, there’s the **none** value which doesn’t render any border at all.

We can also define a different style for each direction of the border, following the same order rules as margin and padding (top, right, bottom then left):

# border-style: dashed dotted outset ridge;

This will render a border where the top is dashed, right is dotted, bottom is outset & left is ridged.

Border can also be used as a shorthand property to define border width, color and style in one declaration:

# border: thin #E0E4CC dotted;

### CSS Media Queries

Media queries allows CSS to identify & react to changes in the browser. Changes such as resizing the window, print preview or switching to a screen reader.

There are lots of media types, like monitors, screen reader and printers. Media queries can identify these media types and apply CSS styling appropriately. Let’s look at an example:

# @media screen {

# body {

# background-color: red;

# }

# }

This media query only applies while the user is viewing the page through a screen (desktop, laptop, mobile etc.).

If the user attempts to print the page, they’ll probably want a white background for visibility:

# @media print {

# body {

# background-color: white;

# }

# }

This changes the background color of the page from red to white only while the user is in print view.

Say the user is using a screen reader to read the page aloud, but we’ve got ads that will confuse it:

# @media speech {

# aside {

# display: none;

# }

# }

The above CSS code will tell the ads in our aside element not to render, allowing the screen reader to work properly.

Using **media height** and **media width** queries, we can ensure that our page content renders appropriately for the aspect ratio of the device the user is viewing from:

# @media (min-width: 500px) {

# nav {

# width: 100px;

# }

# }

This query sets the nav element width to 100 pixels only if the entire browser is 500 or more pixels wide.

The same can be done for height:

# @media (min-height: 500px) {

# nav {

# height: 300px;

# }

# }

We can also do the opposite, for browsers whose width or height is under 500 pixels:

# @media (max-width: 500px) {

# nav {

# width: 50px;

# }

# }

# @media (max-height: 500px) {

# nav {

# height: 50px;

# }

# }

Media queries can understand simple logic using the **and** and **or** logical operators:

# @media (min-width: 500px) and (max-width: 1000px) {

# header {

# font-size: larger;

# }

# }

This code says if the user’s browser width is between 500 and 1000 pixels, make the font size larger.

We can use a comma to represent the **or** logical operator:

# @media (min-width: 1000px), (max-width: 500px) {

# footer {

# font-weight: bold;

# }

# }

We can even combine multiple logical operators to create complex rules:

# @media screen and (min-width: 500px) and (max-width: 500px) {

# article {

# font-style: italic;

# }

# }

### Additional Media Query Features

We can use media queries in <link> elements. This enables us to separate our stylesheets for each platform a user can view our page from:

# <head>

# <link rel="stylesheet" href="base.css" />

# <link rel="stylesheet" href="mobile.css" media="screen and (max-width: 1000px)" />

# <link rel="stylesheet" href="print.css" media="print" />

# </head>

In this example, we’ve used a media query with 3 stylesheets. We use the **media** attribute to define the media type which we want to assign he appropriate stylesheet to. The mobile stylesheet is only called if the browser width is >= 1000px.

There are a few more logical operators we can use with media queries in CSS:

# @media all {

# body > article {

# font-family: serif;

# }

# }

The **all** operator applies the CSS within to all media types

# @media speech {

# body > section.screenReaderOverview {

# display: block;

# }

# }

# @media not speech {

# body > section.screenReaderOverview {

# display: none;

# }

# }

The **not** operator inverses logic (similar to the ‘!’ prefix in Java). Essentially says apply this to all media types except this one.

# <head>

# <link rel="stylesheet" href="base.css" />

# <link rel="stylesheet" href="print.css" media="only print" />

# </head>

Finally, the **only** keyword in HTML is used to tell legacy browsers to ignore the rest of the media query, as older browsers expect a comma-delimited list of media types for *media* attributes in a <link> element.

## Fonts

In this section, we’ll look at how we can use CSS to manipulate the font content within our HTML documents.

### CSS Font Properties

In CSS, we use the **font-family** property to define different fonts that will be used on the HTML page:

# article {

# font-family: Segoe;

# }

# header {

# font-family: Times;

# }

We can use multiple fonts in one declaration, just in case aren’t sure which fonts are installed on the browser:

# article {

# font-family: "Palatino Linotype", "Book Antiqua", Palatino;

# }

# header {

# font-family: "Lucida Sans Unicode", "Lucida Grande", Arial;

# }

The browser will attempt to load the fonts from left to right until it finds one that exists.

Each browser has a default font for specific font types, including sans-serif, serif and monospace. We can use the keyword for these fonts as a failsafe:

# article {

# font-family: "Segoe UI", Segoe, sans-serif;

# }

# header {

# font-family: "Times New Roman", Times, serif;

# }

### Linking to External Font Files

We can use the **@font-face** style declaration in CSS to import an external font file:

# @font-face {

# font-family: "Demo Font";

# src: url(/demofont.otf);

# }

There are a few file formats we can use when importing fonts:

* OpenType Font (.OTF)
* TrueType Font (.TTF)
* Web Open Font Format (.WOFF)
* Web Open Font Format 2.0 (.WOFF2) – not supported in IE
* Embedded OpenType Fonts (.EOT) – *only* supported in IE
* SVG Fonts (.SVG) – only supported in Chrome and Opera

Now that we’ve defined the font-family we want, we can use it in CSS styles:

# footer {

# font-family: "Demo Font", Arial, sans-serif;

# }

We can also define other properties of a font with the @font-face style declaration, such as bold or italics:

# @font-face {

# font-family: "Simple Font";

# src: url(simplefont.ttf);

# }

# @font-face {

# font-family: "Simple Font";

# font-weight: bold;

# src: url(simplefont\_bold.ttf);

# }

# @font-face {

# font-family: "Simple Font";

# font-style: italic;

# src: url(simplefont\_italic.ttf);

# }

Once we define the file for each property, we can use them like normal throughout the rest of the document:

# body {

# font-family: "Simple Font";

# }

# footer {

# font-style: italic;

# }

# header {

# font-weight: bold;

# }

## Module 4: Intro to JavaScript

In this module, we’ll learn some basics of JavaScript. We will examine why it’s useful and how to use it in conjunction with HTML and CSS.

## Why JavaScript?

**JavaScript** is a high level, client side object-oriented scripting language used to create interactive effects within a web browser, and can be used to do many things like:

* Adding multimedia elements (I.E.: showing, hiding, creating or changing image rollovers; scrolling text across status bar, etc.)
* Create tailored dynamic page content, date and time or other external data
* Form processing, such as form validation upon user submission or modifying contents of the form

JS can also be used with CSS to make Dynamic Hypertext Markup Language (DHTML). JS executes only when the JS app is in view in the browser window. When you click over to a different tab, the script stops executing. Client-sided storage API’s and cookies being used by multiple pages which transfer information between each other are the only exceptions to this rule. Cookies can persist even after you close the page.

JS statements are written with a **<script>** tag, which lets the browser know that JavaScript code begins here. Scripts are embedded in HTML & executed by the browser. JS relies on browsers to run code, but we can use JS elsewhere.

### Adding JS to an HTML page

Now that we know how to implement JS in an HTML doc, let’s practice by creating a JS app that modifies text & reacts to clicking a button. We’ll start by adding an id attribute to a header element so our JS has a reference point:

# <header id="headerItem">Old Header</header>

Add our clickable button:

# <button>Click Here</button>

Then we’ll add a JS function that utilizes the **getElementById** method to identify which element we want to modify:

# <script type="text/javascript">

# function startLogic() {

# var newHtml = '<em>New Header</em>';

# document.getElementById('headerItem').innerHTML = newHtml;

# }

# </script>

We’ve added our function, now we need to call the function. To do this, we will write the name of our function within the script element (but outside of the declaration):

# startLogic();

Still within the script element, add the button click function:

# function clickHandler() {

# alert('You clicked me!');

# }

We need to call this function too, but this one’s a bit different. Since we want this code to execute when the user clicks the button, we will call it with the **onclick** attribute inside of the button element:

# <button onclick="clickHandler()">Click Here</button>

The end result looks like this:

# <!DOCTYPE etc...>

# <header id="headerItem">Old Header</header>

# <section>

# <button onclick="clickHandler()">Click Here</button>

# </section>

# <script type="text/javascript">

# function startLogic() {

# var newHtml = '<em>New Header</em>';

# document.getElementById('headerItem').innerHTML = newHtml;

# }

# startLogic();

# function clickHandler() {

# alert('You clicked me!');

# }

# </script>

This code manipulates the content of the element with the ‘headerItem’ ID & sends an alert that says “You clicked me!” when the button is clicked.

### Basic Expressions

In most browsers, we can activate the developer tools console by pressing F12. With these tools, we can perform many useful tasks like view the Document Object Model (DOM), debug JavaScript, view performance stats and much more. We will be using the ‘Console’ tab to test JavaScript.

In JS, we use the **var** keyword to create variables, then we assign a name and a value:

# var firstNumber = 1;

but if we type this alone into our console, it will attempt to evaluate the expression, and return ‘undefined’. This is because the variable firstNumber *was* undefined when we typed that line of code, but now it has been assigned a value of 1. Now if we type firstNumber into our console, it will return 1. If we create a second variable named secondNumber and assigned it a value of 2, we can combine our two values to calculate basic math functions:

# firstNumber + secondNumber

will return a value of 3, since 1 + 2 = 3.

an important differentiation in JS is the use of ‘=’ vs. ‘==’. A single equal sign is used to assign a value, while two equal signs is a Boolean operator. That means it does NOT assign value, but rather evaluates both sides of the expression to return a true or false value. For example, we assigned firstNumber a value of 1, so this expression:

# firstNumber == 2;

equates to **false**, as the value of firstNumber is NOT equal to 2.

Once we’ve assigned a numeric value to a couple of variables, we can use those variables themselves to define a third variable:

# var thirdNumber = firstNumber + secondNumber;

Now thirdNumber has been assigned a dynamic value of whatever the values of firstNumber plus secondNumber equate to. I say dynamic because the value of thirdNumber will change with the other values. If we set a loop to add 1 to firstNumber each iteration, the value of firstNumber will change, and thirdNumber will change with it.

We can also call a variety of browser functions with the developer’s console:

# window.alert(‘Hello, World!’);

This will make a dialogue appear that says “Hello, World!”.

or:

console.log(‘test’);

this logs information to the console and displays the eval code.

We can even write HTML to the page using the DOM using ‘**document.write**’:

# document.write(‘<h1>Header</h1><p>Test</p><footer>Footer</footer>’);

### Operators

We briefly discussed operators before, but let’s jump into some definitions now:

**=** assigns value to a variable

+= assigns an amount to be added to a value

-= assigns an amount to be subtracted from a value

\*= assigns an amount to multiply the value by

/= assigns an amount to divide a value by

%= modulus operator. Not really sure tbh. square root?

== tests two values for equality (Boolean value)

!= tests two values for Inequality (also Boolean)

=== compares two values for equal values and types (type being integer, string etc) whereas == only compares value

!== compares two values and types to see if they’re NOT equal

++ increment

-- decrement

+ adds numeric values together, but can also be used to concatenate multiple string values. Can also use += to affix a new string to the end an existing one:

# var greeting = "Happy ";

# greeting += "Holidays"; // greeting is now "Happy Holidays"

> greater than

< less than

>= greater than or equal to

<= less than or equal to

? ternary operator

There are also operators that infer information based on type or value:

**typeof** returns the type of a variable

**instanceof** returns true if an object is an instance of an object type

### Control Statements

Programs generally perform actions based on the result of different decisions it had to make to get to that action. The statements that pick a path of execution based on decisions are called **control statements**.

an **if** statement runs code within its block if the statement within the parentheses evaluates to true:

# if (day > 5) {

# greeting = "Have a good weekend!";

# }

In the code above, if the value of **day** is greater than 5, then greeting will be assigned the string value; but what if we want it to execute alternate code if the statement equates to false? That’s where **else** comes into play:

# if (day > 5) {

# greeting = "Have a good weekend!";

# } else {

# greeting = "Welcome to work";

# }

This allows us to provide alternate code to execute when different conditions exist.

The **else if** statement combines multiple if statements to evaluate multiple conditions. The last else statement in the block is only executed if all others evaluate to false:

# if (day > 5) {

# greeting = "Have a good weekend!";

# } else if (day < 2) {

# greeting = "Are you having a case of the Mondays!";

# } else {

# greeting = "Welcome to work";

# }

Let’s talk about loops. Loops do exactly what you’d think - execute a block of code over and over until a criterion is met.

The **for** loop is made up of 3 parts, each separated by a semicolon:

1. the variable we’ll be working with and the value we want to assign it (i = 0 in the example below)
2. the Boolean expression the loop evaluates. Loop will iterate until this evaluates to false.
3. the code we want to execute after the the previous code in this block has been executed (i++ below)

# var countries = ["USA", "JPN", "RUS", "ENG"];

# for (var i = 0; i < countries.length; i++) {

# text += countries + "<br />";

# }

The **foreach**() method allows us to run a function on each item in an array:

# var sum = 0;

# var numbers = [4, 9, 16, 25];

# function increment(item, index) {

# sum += item;

# }

# numbers.forEach(increment);

# alert(sum);

The **switch** statement gets a value within parentheses (expression to evaluate) and compares it to as many case values as you have set. It works from top to bottom, and only executes case code when it identifies it as match to the value in parentheses. When a match is identified, all code within the switch statement from that point on will execute until a break is reached. Breaks are used to separate cases:

# var day;

# switch (new Date().getDay()) {

# case 0:

# day = "Sunday";

# break;

# case 1:

# day = "Monday";

# break;

# case 2:

# day = "Tuesday";

# break;

# case 3:

# day = "Wednesday";

# break;

# case 4:

# day = "Thursday";

# break;

# case 5:

# day = "Friday";

# break;

# case 6:

# day = "Saturday";

# break;

# }

# alert(day);

## JavaScript Types

Each variable in JavaScript contains a data type which dictates the value that can be stored in it. In this section, we’ll learn about each type of data we can use in JS and how we can utilize them.

### Primitive Data Types

Variable data types are automatically assigned based on the data within the variable (ex: var word = ‘hello’ is string; var num = 45 is integer) Primitive data types are simple in that they hold no additional properties or methods. These data types are:

* Number
* Undefined
* Boolean
* Null
* String

The **Boolean** data type can either be true or false. Usually used to represent the outcome of comparisons.

The **Number** data type represents all numeric data. Positives, negatives, integers, real numbers, scientific notation, hexadecimals and octals.

The **String** data type represents textual data, and is created by closing a character sequence in double or single quotes.

### Composite Data Types (Objects)

An **object** is a list of unorganized primitive data types stored as a sequence of name-value pairs. Each item in this list is considered a **property** of the object. When an object stores a function, it becomes a **method**.

To refer to an object property, we use the following syntax: **Object.Property** where Object is the object and Property is the property of the object. Objects can also serve as **Associative Arrays**, which are normal arrays but are indexed with strings as opposed to numbers.

A programming language qualifies as ‘object-oriented’ when it has four basic capabilities:

1. **Encapsulation** – bundling of data with methods operating on the data
2. **Aggregation** – storing one object in another
3. **Inheritance** – the ability of a class to inherit properties and methods from other classes
4. **Polymorphism** – a language’s ability to process objects differently based on data type or class

There are a few things we should keep in mind when working with JavaScript objects:

**Object Properties** – normal variables used in the internal object’s methods – may be globally visible variables.

**Object Methods** – functions that enable objects to perform an action or have one performed on it – methods are responsible for executing mathematical operations

**User-Defined Objects** – user-defined and built-in objects are descendants of an object called Object

**‘this’** **Keyword** – used to reference shorthand for an object’s properties or methods

**JavaScript Native Objects** – JS has many built-in, or **native** objects. These objects are accessible anywhere in your program and will work the same way in any browser/OS

Here are some of the major built-in object in JS:

* JavaScript Number Object
* JavaScript Boolean Object
* JavaScript String Object
* JavaScript Array Object
* JavaScript Date Object
* JavaScript Math Object
* JavaScript RegExp Object

### Collections

JavaScript arrays can be used to collect multiple values and store them in a single variable. Arrays are created using the square brackets, ‘[ ]’.

# var cars = [“BMW”, “Porsche”, “Mercedes”];

Above is an array named cars. To reference an item in an array, we refer to its position in the array called its **index**. Array indexes are zero based, meaning the first item is indexed at 0, the second is 1 and so on. We can select items in an array like so:

# cars[0] //BMW

# cars[1] //Porsche

# cars[2] //Mercedes

We can also create an array using the **Array** keyword in JS:

# var cars = new Array(“BMW”, “Porsche”, “Mercedes”);

Arrays can contain values of various types, combining numbers, strings, functions and even another array within one array:

# var arrayA = [];

# myArray[0] = Date.now;

# myArray[1] = myFunc;

# myArray[2] = new Array("Demo", "Test");

### Undefined and Null

JS includes two possible states for a variable if it doesn’t have a value assigned: **undefined** and **null**.

The **undefined** data type represents a value that doesn’t exist, or is not known. This type consists of a single value – undefined. Variables whose values have not been defined are defaulted to undefined. You can also intentionally assign the undefined value to a variable.

The **null** data type is similar to undefined – represented by a single inexistent value “null”, but its absence of value is usually used to initialize, or clear, object variables.

## JavaScript Callbacks