

UC BERKELEY EXTENSION

Database/Application/Programming Courses

Instructor: Michael Kremer, Ph.D.



Course Title: Introduction to Full-Stack Web Development

Course Subtitle: Node.js and the MEAN Stack

Course Number: X443.1

Introduction to Full-Stack Web Development

Instructor: Michael Kremer, Ph.D.

E-mail: mkremer@berkeley.edu

Copyright © 2022

All rights reserved. This publication, or any part thereof, may not be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, storing, or otherwise without expressed permission.

2nd Edition

ICON KEY



Note



Hands-on Example



Demo Example



Warning

TABLE OF CONTENT

CLASS 1 1

1. INTRODUCTION TO FULL STACK WEB DEVELOPMENT	1
1.1 What is a Full Stack?	1
1.2 The MEAN Stack?	4
1.3 Installing of software components	5
2. BASICS OF HTML	6
2.1 Overview of HTML	6
Understanding Common HTML Terms	6
2.2 HTML Tags	7
Elements	7
Tags	7
Attributes	8
2.3 HTML Document Structure	9
Code Validation	11
2.4 HTML Basic Elements	13
Divisions & Spans	13
Text-Based Elements	16
2.5 Hyperlinks	20
Relative & Absolute Paths	20
Linking to an Email Address	23
Opening Links in a New Window	24
Linking to Parts of the Same Page	24
2.6 HTML Form Elements	25
Initializing a Form	25
Text Fields & Textareas	26
Multiple Choice Inputs & Menus	29
Form Buttons	33
Other Inputs	34
2.7 Organizing Form Elements	35
Label	35
Fieldset	36
Legend	36
2.8 Form & Input Attributes	37
Disabled	37
Placeholder	37
Required	38
Additional Attributes	38
3. CASCADING STYLE SHEETS (CSS)	40
3.1 Overview of CSS	40
Understanding Common CSS Terms	40

3.2 CSS Style Rules	41
Selectors.....	41
Properties.....	42
Values.....	42
3.3 Working with Selectors.....	43
Type Selectors	43
Class Selectors.....	43
ID Selectors	44
Additional Selectors	44
3.4 Referencing CSS	45
3.5 Using CSS Resets	48
3.6 The Cascade	49
Cascading Properties.....	49
3.7 Calculating Specificity	50
3.8 Combining Selectors	52
Specificity Within Combined Selectors	53
Layering Styles with Multiple Classes.....	55

CLASS 2 57

4. JAVASCRIPT (ECMAScript)	57
4.1 Overview of JavaScript.....	57
What Can In-Browser JavaScript Do?	58
The “Script” Tag	59
4.2 Basic Language Elements	60
Statements.....	60
Semicolons	60
Comments.....	62
4.3 Variables	63
Variable naming	64
4.4 Data Types and Type Conversions	66
Number	67
String.....	70
Boolean	73
Null	73
Undefined	74
Objects	Error! Bookmark not defined.
4.5 Operators and Expressions	75
String Concatenation.....	76
Assignment.....	77
Increment/Decrement	78
Conditional Operators.....	80
Comparisons	84
Strict Equality	86
Logical Operators	87
4.6 Functions.....	109
Function Declaration	109

Local Variables	110
Outer Variables	110
Global Variables	111
Parameters.....	111
Default Values	112
Evaluation of Default Parameters	113
Returning a Value.....	113

CLASS 3..... 97

5. JQUERY.....	ERROR! BOOKMARK NOT DEFINED.
5.1 What is jQuery?	124
What are the advantages of learning jQuery?	124
Getting Started with jQuery	125
jQuery Object	125
5.2 jQuery Selectors	126
This Selection Keyword	126
jQuery Selection Filters	126
5.3 Traversing	129
Chaining Methods	129
Traversing Methods	130
5.4 Manipulation	133
Getting & Setting.....	133
Attribute Manipulation	133
Style Manipulation	134
DOM Manipulation	137
5.5 Events.....	139
Event Flexibility	139
Nesting Events.....	140
Event Methods.....	140
5.6 Effects	143
jQuery CSS Animations.....	143
Effect Duration	143
Effect Easing.....	144
jQuery UI	144
Effect Callback.....	144
6. WEB PAGES	146
6.1 Elements of a Web Page?	146
6.2 Building a Simple Web Page	148
6.3 Building a Responsive Web Page.....	150
7. INTRODUCTION TO NODE	152
7.1 JavaScript and Node	152
7.2 Advantages of Node.js.....	154
Performance	154
Uses JavaScript.....	154
Node Package Manager (NPM)	154
Cross-Platform	154

Open-Source	155
Lightweight	155
Asynchronous.....	155
Freedom.....	155
7.3 Basic Node.js Architecture	156
7.4 Node.js Library	159
7.5 Node.js Process Model.....	160

CLASS 4 152

8. NODE PROGRAMMING FUNDAMENTALS	162
8.1 Node.js REPL	162
8.2 Node.js Basics	165
Event Loop	165
Buffer	168
Streams	174
Process object	185
Scope in Node	186
8.3 Node.js Modules	188
Node.js Core Modules.....	188
Node.js Local Module.....	190
Export Literals	192
Export Object:	194
Export Function	196
Export function as a Class:	197
8.4 Node Package Manager (NPM).....	198
Install Package Locally	198
Add Dependency into package.json:.....	199
Install Package Globally:.....	200
Update Package	200
Uninstall Packages.....	200
8.5 Asynchronous Programming Techniques	201
8.6 Debugging Node.js Code.....	208
Core Node Debugger.....	208
Node Inspector.....	210

CLASS 5 212

9. REQUEST/RESPONSE OBJECTS	212
9.1 HTTP Protocol	212
9.2 URL Decomposition.....	213
9.3 Request Methods.....	215
9.4 Request Headers.....	217
9.5 Response Headers.....	219
9.6 Internet Media Types.....	222
9.7 Request Body	224

9.8 Request Object.....	225
9.9 Response Object.....	227
10. BUILDING NODE WEB APPLICATIONS	229
10.1 Simple HTTP Server	229
10.2 Building RESTful Web Services.....	234
Post	234
GET	238
Cross-Origin Resource Sharing (CORS)	241
DELETE	242
PUT	245
10.3 Serving Static Files	248
Optimizing Data Transfer using Pipe	250
Handling server errors.....	252
10.4 Accepting User Inputs from Forms	255
Handling File Uploads.....	259

CLASS 6 268

11. GETTING STARTED WITH EXPRESS.....	268
11.1 Setting Up an Express Project (Scaffolding).....	268
11.2 Configuring Express/Application	269
11.3 Express App File Structure	272
12. BUILDING EXPRESS APPLICATIONS	279
12.1 Introduction to Templating.....	279
Choosing a Template Engine	281
12.2 Using Handlebars.....	282
Comments.....	283
12.3 Views and Layouts	286
12.4 Rendering Views	289
12.5 Partial.....	294
12.6 Static Files and Views.....	297
12.7 Handling Forms.....	302
Sending Client Data to Server	302
HTML Forms	303
Encoding.....	304
Approaches to Form Handling	304
Form Handling with Express.....	307
12.8 Execution Environments	311

CLASS 7 313

13. ADVANCED TEMPLATING.....	313
13.1 Blocks	313

13.2 Server- vs. Client-Side Templating	318
Server-Side Rendering	319
Client-Side Rendering	319
Pros and Cons (Server-Side, Client-Side)	320
13.3 Custom Block Helpers	323
13.4 Perfecting Your Templates	327
14. PERSISTENCE	328
14.1 Filesystem Persistence	328
14.2 Cloud Persistence	330
14.3 Database Persistence	331
15. DATABASE TECHNOLOGIES	332
15.1 Introduction to Database Technologies	332
The Origins	333
Distributed Systems	334
Main Differences Between Relational and Non-Relational Databases	335
15.2 Relational Databases	336
15.3 NoSQL Databases	337
15.4 Setting Up MongoDB	338
Mongoose	339
Creating Data	342
Retrieving Data	346
Adding/Updating Data	349

CLASS 8 355

16. ADVANCED EXPRESS, PART 1	355
16.1 Cookies	355
Cookies in Express	359
Signed Cookies	363
Examining Cookies	364
16.2 Sessions	367
Express-Session Package	368
Using Sessions	369
16.3 Advanced Routing	375
Routes and SEO	378
Subdomains	379
Route Handlers Are Middleware	380
Route Paths and Regular Expressions	383
Route Parameters	384
Organizing Routes	385
Declaring Routes in a Module	386
Grouping Handlers Logically	387
Other Approaches to Route Organization	389
16.4 Automatic Rendering of Views	390

CLASS 9 393

17. ADVANCED EXPRESS, PART 2	393
17.1 <i>Authenticating Users</i>	393
Authentication Versus Authorization	394
The Problem with Passwords	395
Building Simple Authentication/Authorization App	396
Third-Party Authentication	400
Storing Users in Your Database	402
Passport	403
Authentication Strategies	404
Building an Authentication Application.....	405
17.2 <i>REST APIs and JSON</i>	406
JSON and XML	407
Building the API	408
API Error Reporting	409
18. INTRODUCTION TO ANGULAR	413
18.1 <i>Overview of AngularJS</i>	415
What is AngularJS?	415
Complete client-side solution	416
AngularJS's sweet spot	416
Fundamental Beliefs of AngularJS	417
18.2 <i>Basics of Data Binding</i>	419
ng-app	421
ng-init	422
ng-model	422
18.3 <i>UI Logic: Controllers</i>	424
18.4 <i>Independent Business Logic: Services</i>	428
18.5 <i>Backend Communication</i>	434
18.6 <i>Modules</i>	437
Module Loading	441

CLASS 10 443

19. ADVANCED ANGULARJS.....	443
19.1 <i>Scopes</i>	443
Scope as Data-Model	444
Scope Hierarchies.....	447
Scopes and Directives	452
Directives that Create Scopes	452
Controllers and Scopes.....	452
19.2 <i>Templates</i>	453
Using Filters in View Templates	454
19.3 <i>Data Binding</i>	455
Data Binding in Classical Template Systems.....	455
Data Binding in AngularJS Templates.....	456

19.4 Controllers.....	457
Setting up Initial State of \$scope Object.....	458
Adding Behavior to Scope Object.....	459
19.5 Services	464
Using Services	464
19.6 Using Handlebars and Angular.....	466

CLASS 1

1. Introduction to Full Stack Web Development

1.1 What is a Full Stack?

A stack refers to all of the components needed to run an application. This includes frameworks and components internal to an application, such as UI frameworks, MVC frameworks, data access libraries, etc. It can also include things external to an application, such as the operating system, application server, and database.

A full-stack web developer is a developer that has general knowledge across a wide breadth of technologies and platforms as well as in-depth experience and specialization in a couple of those concepts. For the most part, there are two general fields that make up a full-stack developer's skillset: front-end development and back-end development.

Front-End Development

This skillset involves the actual presentation of your website—how the information in your website is laid out in browsers and on mobile devices as well. A dedicated front-end developer will be very experienced working with HTML and CSS as well as the scripting language, JavaScript. With these languages, the developer can very efficiently manipulate the information on a website to make it appealing and effective.

Everything that you actually see on a website—the layout, the positioning of text and images, colors, fonts, buttons, and so on—are all factors that the front-end developer must consider.

The main goal of a front-end developer is to provide the platform for visitors to interact with, a platform which provides and receives information. This means some developers will be well-versed in web design and using software such as Photoshop and Illustrator to create graphics and themed layouts.

Additional skillsets of a front-end developer could include user experience design and user interface design, skills which help a team evaluate the best methods of displaying and collecting information. A front-end developer who possesses these design skills is potentially more valuable as they can identify the look and feel of a site while assessing the technical capabilities of such a design at the same time.

Back-End Development

Create, edit/update and recollection of data are some of the processes that are most often associated with back-end development. Some examples of common scripting languages used are JavaScript, PHP, Ruby, and Python. With these languages, a back-end developer can create algorithms and business logic to manipulate the data that was received in front-end development.

This means that a back-end developer must be able to write code to receive the information input from the user and also save it somewhere – like in a database.

There are two main types of databases:

- Relational (Oracle, SQL Server, PostgreSQL, MySQL)
- Non-relational (Mongo, Oracle, Couchbase)

The language used for database management is SQL, which helps the developer interact with the database.

The concepts might sound foreign, but just understand that there are different database management systems based on convenience and use.

Another component of back-end development is server management, which are applications that host the database and serve up the website. An alternative to knowing how to manage servers is to use cloud-based platforms that provide the infrastructure, like Heroku or Amazon Web Services.

Understanding server management allows a developer to troubleshoot slow applications and even determine how scalable their websites are to include more users.

Frameworks

Rather than having to develop complex proprietary code every time for creating different websites, frameworks have become popular resources to make many processes more efficient and convenient.

Libraries like jQuery are extremely popular for front-end developers using Javascript, as they can implement various functions that other developers have already cultivated and tested.

Javascript frameworks like AngularJS and EmberJS solve many of the challenges faced by front-end developers by developing conventions that can easily be implemented with any website.

On the backend, there are frameworks like Express for Node.js (JavaScript), Rails for the programming language of Ruby, Django for Python, and CakePHP for working with PHP.

The main purpose of frameworks is to make a developer's job easier by developing a set of conventions that can be adopted for many of the different processes involved in creating a website—from how information is displayed to how it is stored and accessed in the database.

1.2 The MEAN Stack?

The term MEAN stack refers to a collection of JavaScript based technologies used to develop web applications. MEAN is an acronym for

- MongoDB
- Express
- AngularJS
- Node.js

From client to server to database, MEAN is full stack JavaScript.

Node.js is a server-side JavaScript execution environment. It is a platform built on Google Chrome's V8 JavaScript runtime. It helps in building highly scalable and concurrent applications rapidly.

Express is lightweight framework used to build web applications in Node. It provides a number of robust features for building single and multi-page web applications. Express is inspired by the popular Ruby framework, Sinatra.

MongoDB is a schema-less NoSQL database system. MongoDB saves data in binary JSON format which makes it easier to pass data between client and server.

AngularJS is a JavaScript framework developed by Google. It provides some awesome features like the two-way data binding. It is a complete solution for rapid and powerful front-end development.

1.3 Installing of software components

Node.js

To install Node, you simply have to download and unzip the files into a folder of your choice. Then update the path system variable to point to the main folder. See the installation document in Canvas.

NPM:

Node Package Manager (npm) comes automatically with the node installation, however, it may be outdated. But there should be no immediate need to update it, but feel free to update npm to the latest version.

Code Editor:

Simply put, you just need a text editor. I will use Notepad++ in class, but feel free to choose any good text editor of your choice. And you may use an Integrated Development Environment (IDE), such as Atom, Visual Studio, etc.

MongoDB:

I am using a cloud, free instance of MongoDB at [mongodb.com](https://www.mongodb.com) (<https://www.mongodb.com/cloud/atlas>). So, I recommend to create a free account at MongoDB Atlas.

You can also download MongoDB and install it locally, if you wish.

Browser:

I will be using Chrome, Firefox, and Edge in class. Again, you can use any browser of your choice, in general.

2. Basics of HTML

2.1 Overview of HTML

If you can, imagine a time before the invention of the Internet. Websites did not exist, and books, printed on paper and tightly bound, were your primary source of information. It took a considerable amount of effort—and reading—to track down the exact piece of information you were after.

Today you can open a web browser, jump over to your search engine of choice, and search away. Any bit of imaginable information rests at your fingertips. And chances are someone somewhere has built a website with your exact search in mind.

HTML, **H**yper**T**ext **M**arkup **L**anguage, gives content structure and meaning by defining that content as, for example, headings, paragraphs, or images.

CSS, or Cascading Style Sheets, is a presentation language created to style the appearance of content—using, for example, fonts or colors.

The two languages—HTML and CSS—are independent of one another and should remain that way. CSS should not be written inside of an HTML document and vice versa. As a rule, HTML will always represent content, and CSS will always represent the appearance of that content.

Understanding Common HTML Terms

While getting started with HTML, you will likely encounter new—and often strange—terms. Over time you will become more and more familiar with all of them, but the three common HTML terms you should be familiar with are:

- Elements
- Tags
- Attributes

2.2 HTML Tags

Elements

Elements are designators that define the structure and content of objects within a page. Some of the more frequently used elements include multiple levels of headings (identified as h1 through h6 elements) and paragraphs (identified as the p element); the list goes on to include the a, div, span, strong, and em elements, and many more.

Elements are identified by the use of less-than and greater-than angle brackets, < >, surrounding the element name. Thus, an element will look like the following:

Code Example:

```
<a>
```

Tags

The use of less-than and greater-than angle brackets surrounding an element creates what is known as a tag. Tags most commonly occur in pairs of opening and closing tags.

An opening tag marks the beginning of an element. It consists of a less-than sign followed by an element's name, and then ends with a greater-than sign; for example, <div>.

A closing tag marks the end of an element. It consists of a less-than sign followed by a forward slash and the element's name, and then ends with a greater-than sign; for example, </div>.

The content that falls between the opening and closing tags is the content of that element, if applicable. An anchor link, for example, will have an opening tag of <a> and a closing tag of . What falls between these two tags will be the content of the anchor link.

So, anchor tags will look a bit like this:

Code Example:

```
<a>.....</a>
```

Attributes

Attributes are properties used to provide additional information about an element. The most common attributes include:

- id attribute: which identifies an element
- class attribute, which classifies an element
- src attribute, which specifies a source for embeddable content
- href attribute, which provides a hyperlink reference to a linked resource

Attributes are defined within the opening tag, after an element's name. Generally, attributes include a name and a value. The format for these attributes consists of the attribute name followed by an equal sign and then a quoted attribute value.

Syntax:

```
< tag_name attribute_name = "value" >
```

For example, an <a> element including an href attribute would look like the following:

Code Example:

```
<a href="https://www.google.com/">Google</a>
```

The preceding code will display the text "Google" on the web page and will take users to <https://www.google.com/> upon clicking the "Google" text.

The anchor element is declared with the opening <a> and closing tags encompassing the text, and the hyperlink reference attribute and value are declared with href="https://www.google.com/" in the opening tag.

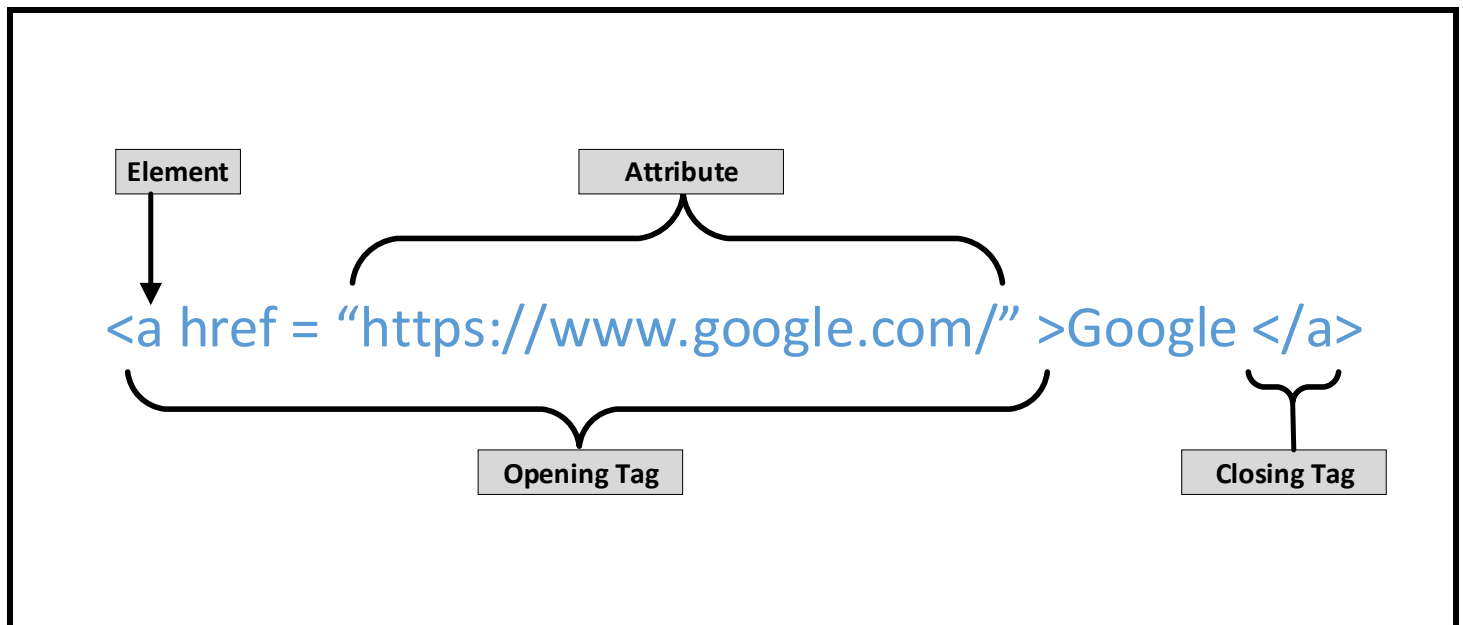


Figure 1: HTML Tag Syntax

2.3 HTML Document Structure

HTML documents are plain text documents saved with an .html file extension rather than a .txt file extension. To begin writing HTML, you first need a plain text editor that you are comfortable using. Sadly, this does not include Microsoft Word or Pages, as those are rich text editors. Two of the more popular plain text editors for writing HTML and CSS are Notepad++ (Windows) and Sublime Text (Mac).

All HTML documents have a required structure that includes the following declaration and elements:

- `<!DOCTYPE html>`
- `<html>`
- `<head>`
- `<body>`

The document type declaration, or `<!DOCTYPE html>`, informs web browsers which version of HTML is being used and is placed at the very beginning of the HTML document. Because we'll be using the latest version of HTML, our document type declaration is simply `<!DOCTYPE html>`. Following the document type declaration, the `<html>` element signifies the beginning of the document.

Inside the `<html>` element, the `<head>` element identifies the top of the document, including any metadata (accompanying information about the page). The content inside the `<head>` element is not displayed on the web page itself. Instead, it may include the document title (which is displayed on the title bar in the browser window), links to any external files, or any other beneficial metadata.

All of the visible content within the web page will fall within the `<body>` element. A breakdown of a typical HTML document structure looks like this:

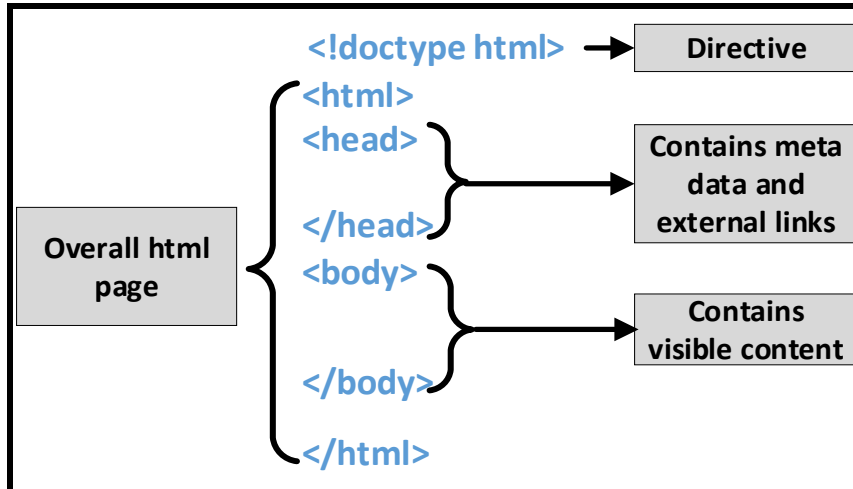


Figure 2: HTML Document Structure

A sample of a simple html page is shown below:

Code Example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello World</h1>
    <p>This is a web page.</p>
  </body>
</html>
```

The preceding code shows the document beginning with the document type declaration, `<!DOCTYPE html>`, followed directly by the `<html>` element. Inside the `<html>` element come the `<head>` and `<body>` elements.


The `<head>` element includes the character encoding of the page via the `<meta charset="utf-8">` tag and the title of the document via the `<title>` element.

The `<body>` element includes a heading via the `<h1>` element and a paragraph via the `<p>` element. Because both the heading and paragraph are nested within the `<body>` element, they are visible on the web page.

When an element is placed inside of another element, also known as nested, it is a good idea to indent that element to keep the document structure well organized and legible. In the previous code, both the `<head>` and `<body>` elements were nested—and indented—inside the `<html>` element. The pattern of indenting for elements continues as new elements are added inside the `<head>` and `<body>` elements.

In the previous example, the `<meta>` element had only one tag and did not include a closing tag. This was intentional! Not all elements consist of opening and closing tags. Some elements simply receive their content or behavior from attributes within a single tag. The `<meta>` element is one of these elements. The content of the previous `<meta>` element is assigned with the use of the `charset` attribute and value. Other common self-closing elements include:

`
` `<embed>` `<hr>` `` `<input>` `<link>` `<meta>` `<param>` `<source>` `<wbr>`

 **Note:** Self-closing elements are also called void elements. They also used to be written with an ending slash, such as `<hr />`

Code Validation

No matter how careful we are when writing our code, we will inevitably make mistakes. Thankfully, when writing HTML and CSS we have validators to check our work. The W3C has built both HTML (<http://validator.w3.org/>) and CSS (<http://jigsaw.w3.org/css-validator/>) validators that will scan code for mistakes.

Validating our code not only helps it render properly across all browsers, but also helps teach us the best practices for writing code.

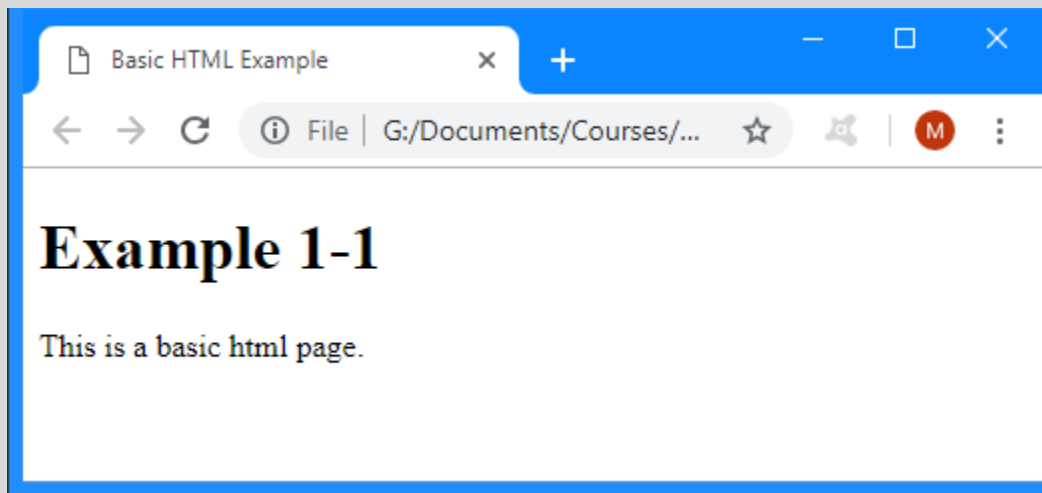
**Example 1-1:** Basic HTML Page

1. Create a file named Example1_1.html containing the following code:

```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Basic HTML Example</title>
6   </head>
7   <body>
8     <h1>Example 1-1</h1>
9     <p>This is a basic html page.</p>
10  </body>
11 </html>
```


Ln: 14 Col: 1 Sel: 0|0 Windows (CR LF) UTF-8 INS

2. Save the file and execute it in a browser.



2.4 HTML Basic Elements

In order to start building websites, we need to learn a little about which HTML elements are best used to display different types of content. It is also important to understand how elements are visually displayed on a web page, as well as what different elements mean semantically.

 **Note:** Semantics within HTML is the practice of giving content on the page meaning and structure by using the proper element. Semantic code describes the value of content on a page, regardless of the style or appearance of that content.

Using the proper element for the job goes a long way, and we will want to make well-informed decisions in the process.

Divisions & Spans

Divisions, or `<div>`s, and ``s are HTML elements that act as containers solely for styling purposes. As generic containers, they do not come with any overarching meaning or semantic value.

Paragraphs are semantic in that content wrapped within a `<p>` element is known and understood as a paragraph. `<div>`s and ``s do not hold any such meaning and are simply containers.

Most elements are either block- or inline-level elements. What is the difference?

- Block-level elements:
 - Begin on a new line, stacking one on top of the other, and occupy any available width
 - May be nested inside one another and may wrap inline-level elements
 - Most commonly block-level elements are used for larger pieces of content, such as paragraphs
- Inline-level elements:
 - Do not begin on a new line
 - Fall into the normal flow of a document, lining up one after the other, and only maintain the width of their content
 - May be nested inside one another; however, they cannot wrap block-level elements
 - Usually inline-level elements are used with smaller pieces of content, such as a few words

**Example 1-2:** Div and Span elements

1. Create a file named Example1_2.html containing the following code:

The screenshot shows a Notepad++ window with the file 'Example1_2.html' open. The code is as follows:

```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Div and Span</title>
6     <!--Make sure to use stylesheet (/* */) comments inside style tag-->
7     <style>/*Stylesheet for div and span elements*/
8       div.intro {
9         border: 1px black solid;
10        background-color: gray;
11        width: 250px;
12      }
13      #spnStrong {
14        font-weight: bold;
15        text-decoration: underline overline;
16      }
17    </style>
18  </head>
19  <body>
20    <h1>Example 1-2</h1>
21    <div class="intro"><!--Div element-->
22      <p>This is a longer section<!--First paragraph-->
23      that needs to be<!--First paragraph-->
24      <span id="spnStrong">formatted</span><!--Span element -->
25      as one unit.</p><!--First paragraph ends-->
26      <p>This is another section</p><!--Second paragraph-->
27    </div><!--Div element ends-->
28  </body>
29 </html>
```

Below the code editor, a preview window titled 'Div and Span' shows the rendered output. It features a large heading 'Example 1-2'. Below it, a gray-bordered box contains the text 'This is a longer section that needs to be **formatted** as one unit.' followed by 'This is another section'.

Both `<div>`s and ``s, however, are extremely valuable when building a website in that they give us the ability to apply targeted styles to a contained set of content:

- `<div>` is a block-level element that is commonly used to identify large groupings of content, and which helps to build a web page's layout and design
- ``, on the other hand, is an inline-level element commonly used to identify smaller groupings of text within a block-level element

We will commonly see `<div>`s and ``s with class or id attributes for styling purposes. Choosing a class or id attribute value, or name, requires a bit of care. We want to choose a value that refers to the content of an element, not necessarily the appearance of an element.

For example, if we have a `<div>` with an orange background that contains social media links, our first thought might be to give the `<div>` a class value of orange. What happens if that orange background is later changed to blue? Having a class value of orange no longer makes sense. A more sensible choice for a class value would be social, as it pertains to the contents of the `<div>`, not the style.

Code Example:

```
<!-- Division -->
<div class="social">
  <p>I may be found on...</p>
  <p>Additionally, I have a profile on...</p>
</div>
<!-- Span -->
<p>Soon we'll be <span class="tooltip">writing HTML</span> with the best of them.</p>
```

Text-Based Elements

Many different forms of media and content exist online; however, text is predominant. Accordingly, there are a number of different elements for displaying text on a web page. For now, we will focus on the more popular elements, including headings, paragraphs, bold text to show importance, and italics for emphasis.

Headings

Headings are block-level elements, and they come in six different rankings, `<h1>` through `<h6>`. Headings help to quickly break up content and establish hierarchy, and they are key identifiers for users reading a page. They also help search engines to index and determine the content on a page.

Headings should be used in an order that is relevant to the content of a page. The primary heading of a page or section should be marked up with an `<h1>` element, and subsequent headings should use `<h2>`, `<h3>`, `<h4>`, `<h5>`, and `<h6>` elements as necessary.

Each heading level should be used where it is semantically valued, and should not be used to make text bold or big—there are other, better ways to do that.

Here is an example of HTML for all the different heading levels and the resulting display on a web page.

Code Example:

```
<h1>Heading Level 1</h1>
<h2>Heading Level 2</h2>
<h3>Heading Level 3</h3>
<h4>Heading Level 4</h4>
<h5>Heading Level 5</h5>
<h6>Heading Level 6</h6>
```

Paragraphs

Headings are often followed by supporting paragraphs. Paragraphs are defined using the `<p>` block-level element. Paragraphs can appear one after the other, adding information to a page as desired.

Bold Text with Strong

To make text bold and place a strong importance on it, we will use the `` inline-level element. There are two elements that will bold text for us: the `` and `` elements. It is important to understand the semantic difference between the two.

The `` element is semantically used to give strong importance to text, and is thus the most popular option for bolding text. The `` element, on the other hand, semantically means to stylistically offset text, which is not always the best choice for text deserving prominent attention. We have to gauge the significance of the text we wish to set as bold and to choose an element accordingly.

Code Example:

```
<!-- Strong importance -->
<p><strong>Caution:</strong> Falling rocks.</p>
<!-- Stylistically offset -->
<p>This recipe calls for <b>bacon</b> and <b>mayonnaise</b>.</p>
```

Italicize Text with Emphasis

To italicize text, thereby placing emphasis on it, we will use the `` inline-level element. As with the elements for bold text, there are two different elements that will italicize text, each with a slightly different semantic meaning.

The `` element is used semantically to place a stressed emphasis on text; it is thus the most popular option for italicizing text. The other option, the `<i>` element, is used semantically to convey text in an alternative voice or tone, almost as if it were placed in quotation marks. Again, we will need to gauge the significance of the text we want to italicize and choose an element accordingly.

Code Example:

```
<!-- Stressed emphasis -->
<p>I <em>love</em> Chicago!</p>
<!-- Alternative voice or tone -->
<p>The name <i>Shay</i> means a gift.</p>
```

Encoding Special Characters

To display special characters and punctuation marks that have special meaning in the html language we need to encode those characters (similar to escape) in order to display them as content.

Reserved characters in HTML must be replaced with character entities. Characters that are not present on your keyboard can also be replaced by entities.

Special characters include various punctuation marks, accented letters, and symbols. When typed directly into HTML, they can be misunderstood or mistaken for the wrong character; thus they need to be encoded.

Each encoded character will begin with an ampersand, &, and end with a semicolon, ;. What falls between the ampersand and semicolon is a character's unique encoding, be it a name or numeric encoding.

Code Example:

```
<p>Peter & Mary ... </p> → displays: Peter & Mary
```

Or

```
<p>Peter &#38; Mary ... </p> → displays: Peter & Mary
```

You can use either the entity name (&) or the entity number (&). In general, an entity name is easy to remember. But some browsers may not support all entity names, but the support for numbers is good.

A common character entity used in HTML is the non-breaking space: A non-breaking space is a space that will not break into a new line.

Two words separated by a non-breaking space will stick together (not break into a new line). This is handy when breaking the words might be disruptive.

Another common use of the non-breaking space is to prevent browsers from truncating spaces in HTML pages. If you write 10 spaces in your text, the browser will remove 9 of them. To add real spaces to your text, you can use the character entity.

**Example 1-3:** Character Entities

1. Create a file named Example1_3.html containing the following code:

The screenshot shows a code editor window titled 'Example1_3.html' with the following HTML code:

```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Character Entities</title>
6   </head>
7   <body>
8     <h1>Example 1-3</h1>
9     <p>An paragraph element is enclosed in
10    opening and closing tags using
11    <p> and </p><!--Interpreted as html paragraph element-->
12    &lt;p&gt; and &lt;/p&gt;<!--Using character entities-->
13    brackets.</p>
14   </body>
15 </html>
```

The status bar at the bottom of the editor indicates 'length : 396 lines : 23'.

Below the editor is a preview window titled 'Character Entities' showing the rendered HTML. The preview displays the title 'Example 1-3' and the paragraph text: 'An paragraph element is enclosed in opening and closing tags using and <p> and </p> brackets.'

**Demo Example 1-1:** HTML Page using Basic Elements

Execute the Demo Example

2.5 Hyperlinks

Along with text, one of the core components of the Internet is the hyperlink, which provides the ability to link from one web page or resource to another. Hyperlinks are established using the anchor, `<a>`, inline-level element.

In order to create a link from one page (or resource) to another, the href attribute, known as a hyperlink reference, is required. The href attribute value identifies the destination of the link.

Relative & Absolute Paths

The two most common types of links are:

- Links to other pages of the same website
- Links to other websites

These links are identified by their href attribute values, also known as their paths.

Links pointing to other pages of the same website will have a relative path, which does not include the domain (.com, .org, .edu, etc.) in the href attribute value. Because the link is pointing to another page on the same website, the href attribute value needs to include only the filename of the page being linked to: about.html, for example.

Should the page being linked to reside within a different directory, or folder, the href attribute value needs to reflect this as well. Say the about.html page resides within the pages directory; the relative path would then be pages/about.html.

Linking to other websites outside of the current one requires an absolute path, where the href attribute value must include the full domain. A link to Google would need the href attribute value of `http://google.com`, starting with `http` and including the domain, `.com` in this case.

Here clicking on the text "About" will open the `about.html` page inside our browser. Clicking the text "Google," on the other hand, will open `http://google.com/` within our browser.

Code Example:

```
<!-- Relative Path -->
<a href="about.html">About</a>
<!-- Absolute Path -->
<a href="http://www.google.com/">Google</a>
```

In general, it is considered best-practice to use relative URLs, so that your website will not be bound to the base URL of where it is currently deployed. For example, it will be able to work on `localhost`, as well as on your public domain, without modifications.

Some examples of relative paths are shown in Table 1.

Path	Description
<code></code>	The " page.html " file is located in the same folder as the current page
<code></code>	The " page.html " file is located in the images folder in the current folder
<code></code>	The " page.html " file is located in the images folder at the root of the current web
<code></code>	The " page.html " file is located in the folder one level up from the current folder

Table 1: Relative Path Examples

**Example 1-4:** Anchor Element

1. Create a file named Example1_4.html containing the following code:

The screenshot displays a code editor with two files open: `Example1_4.html` and `Example1_4_anchor.html`. The `Example1_4.html` file contains the following HTML code:

```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Anchor Element</title>
6   </head>
7   <body>
8     <h1>Example 1-4</h1>
9     <a href="https://google.com">Google</a><!--Absolute Path-->
10    <hr />
11    <!--Change relative path to sub folder pages-->
12    <a href="Example1_4_anchor.html"><!--Relative Path-->
13      Local page relative to current page</a>
14  </body>
15 </html>
```

The `Example1_4_anchor.html` file contains the following HTML code:

```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Local Page</title>
6   </head>
7   <body>
8     <h1>Example 1-4</h1>
9     <p>This is a local page relative
10    to current page.</p>
11    <a href="Example1_4.html">
12      Going back to home</a>
13  </body>
```

Below the code editor, a browser window titled "Anchor Element" shows the rendered output of the first file. The browser address bar displays `file:///F:/Documents/Courses/FullStackWebDevelopment/Scripts/HandsOnExamples_new/Class1/Solution/Example1_4.html`. The page content is as follows:

Example 1-4

[Google](https://google.com)

[Local page relative to current page](#)

Linking to an Email Address

Occasionally we may want to create a hyperlink to our email address—for example, hyperlink text that says “Email Me,” which when clicked opens a user’s default email client and pre-populates part of an email. At a minimum the email address to which the email is being sent is populated; other information such as a subject line and body text may also be included.

To create an email link, the href attribute value needs to start with `mailto:` followed by the email address to which the email should be sent. To create an email link to `mkremer@awesome.com`, for example, the href attribute value would be `mailto:mkremer@awesome.com`.


Additionally, subject, body text, and other information for the email may be populated. To add a subject line, we will include the `subject=` parameter after the email address. The first parameter following the email address must begin with a question mark, `?`, to bind it to the hyperlink path. Multiple words within a subject line require that spaces be encoded using `%20`.

Adding body text works in the same way as adding the subject, this time using the `body=` parameter in the href attribute value. Because we are binding one parameter to another we need to use the ampersand, `&`, to separate the two. As with the subject, spaces must be encoded using `%20`, and line breaks must be encoded using `%0A`.

Altogether, a link to `mkremer@awesome.com` with the subject of “Reaching Out” and body text of “How are you” would require an href attribute value of

Code Example:

```
mailto:mkremer@awesome.com?subject=Reaching%20Out&body=How%20are%20you.
```

 **Warning:** The `mailto` link delegates the use of an e-mail program completely to the client and the underlying OS. Depending on the client setup, the `mailto` link may not work at all!

Opening Links in a New Window

One feature available with hyperlinks is the ability to determine where a link opens when clicked. Typically, links open in the same window from which they are clicked; however, links may also be opened in new windows.

To trigger the action of opening a link in a new window, use the `target` attribute with a value of `_blank`. The `target` attribute determines exactly where the link will be displayed, and the `_blank` value specifies a new window.

Code Example:

```
<a href="https://google.com/" target="_blank">Dr. Google</a>
```

Linking to Parts of the Same Page

Periodically we will see hyperlinks that link to part of the same page the link appears on. A common example of these same-page links are “Back to top” links that return a user to the top of a page.

We can create an on-page link by first setting an `id` attribute on the element we wish to link to, then using the value of that `id` attribute within an anchor element’s `href` attribute.

Using the “Back to top” link as an example, we can place an `id` attribute value of `top` on the `<body>` element. Now we can create an anchor element with an `href` attribute value of `#top`, pound sign and all, to link to the beginning of the `<body>` element.

Code Example:

```
<body id="top">
...
<a href="#top">Back to top</a>
...
</body>
```



Demo Example 1-2: HTML Page Using Anchor Links

Execute the Demo Example

2.6 HTML Form Elements

Forms are an essential part of the Internet, as they provide a way for websites to capture information from users and to process requests, and they offer controls for nearly every imaginable use of an application.

Through controls or fields, forms can request a small amount of information—often a search query or a username and password—or a large amount of information—perhaps shipping and billing information or an entire job application.

We need to know how to build forms in order to acquire user input. In this chapter we will discuss how to use HTML to mark up a form, which elements to use to capture different types of data, and how to style forms with CSS.

We will not get too deep into how information from a form is processed and handled on the back end of a website yet. Form processing is a deeper topic, and we will cover this later in this course using server-side processing.

Initializing a Form

To add a form to a page, we will use the `<form>` element. The `<form>` element identifies where on the page control elements will appear. Additionally, the `<form>` element will wrap all of the elements included within the form, much like a `<div>` element.

Syntax:

```
<form action="action_value" method="method_name">  
...  
</form>
```

A handful of different attributes can be applied to the `<form>` element, the most common of which are action and method:

- The action attribute contains the URL to which information included within the form will be sent for processing by the server
- The method attribute is the HTTP method browsers should use to submit the form data. Both of these `<form>` attributes pertain to submitting and processing data

Text Fields & Textareas

When it comes to gathering text input from users, there are a few different elements available for obtaining data within forms. Specifically, text fields and textareas are used for collecting text- or string-based data. This data may include passages of text content, passwords, telephone numbers, and other information.

Text Fields

One of the primary elements used to obtain text from users is the `<input>` element. The `<input>` element uses the `type` attribute to define what type of information is to be captured within the control. The most popular `type` attribute value is `text`, which denotes a single line of text input.

Along with setting a `type` attribute, it is best practice to give an `<input>` element a `name` attribute as well. The `name` attribute value is used as the name of the control and is submitted along with the input data to the server.

Code Example:

```
<input type="text" name="username">
```

The `<input>` element is self-contained, meaning it uses only one tag and it does not wrap any other content. The value of the element is provided by its attributes and their corresponding values.

Originally, the only two text-based type attribute values were text and password (for password inputs); however, HTML5 brought along a handful of new type attribute values.

These values were added to provide clearer semantic meaning for inputs as well as to provide better controls for users. Should a browser not understand one of these HTML5 type attribute values, it will automatically fall back to the text attribute value. Below is a list of the new HTML5 input types:

- | | | |
|------------|----------|--------|
| ➤ color | ➤ file | ➤ tel |
| ➤ date | ➤ month | ➤ time |
| ➤ datetime | ➤ number | ➤ url |
| ➤ email | ➤ range | ➤ week |
| | ➤ search | |

These <input> elements provide potential different controls and also on mobile devices change the keyboard layout to account for specific keys to enter for this control, such as a phone number would use only number keys.

Text Area

Another element that is used to capture text-based data is the <textarea> element. The <textarea> element differs from the <input> element in that it can accept larger passages of text spanning multiple lines.

The <textarea> element also has start and end tags that can wrap plain text. Because the <textarea> element only accepts one type of value, the type attribute does not apply here, but the name attribute is still used.

The <textarea> element has two sizing attributes:

- cols for width in terms of the average character width
- rows for height in terms of the number of lines of visible text

The size of a textarea, however, is more commonly identified using the width and height properties within CSS.

Code Example:

```
<textarea name="comment" cols="10" rows="5">Add your comment here</textarea>
```

**Example 1-5:** HTML Form Example using Various Text Input and Button

1. Create a file named Example1_5.html containing the following code:

The screenshot shows a Notepad++ window with the following HTML code:

```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Form Input Elements</title>
6   </head>
7   <body>
8     <h1>Example 1-5</h1>
9     <form id="frm" action="#">
10       <!--Type=text is the default-->
11       <p><input type="text" id="txt" placeholder="Enter Text"></p>
12       <!--Input type button-->
13       <p><input type="button" id="btn" value="Click here"></p>
14       <!--Textarea with initially 5 rows and 20 columns -->
15       <p><textarea id="ta" rows="5" cols="20">Enter Text</textarea></p>
16     </form>
17   </body>
18 </html>
```

The browser window displays the rendered form with the title "Example 1-5". It contains a text input field with the placeholder "Enter Text", a button labeled "Click here", and a text area with the placeholder "Enter Text".

2. Execute the file in a browser:

**Demo Example 1-3:** HTML Form Example using Various HTML5 Input Elements

Execute the Demo Example

Multiple Choice Inputs & Menus

Apart from text-based input controls, HTML also allows users to select data using multiple choice and drop-down lists. There are a few different options and elements for these form controls, each of which has distinctive benefits.

Radio Buttons

Radio buttons are an easy way to allow users to make a quick choice from a small list of options. Radio buttons permit users to select one option only, as opposed to multiple options.

To create a radio button, the `<input>` element is used with a `type` attribute value of `radio`. Each radio button element should have the same `name` attribute value so that all of the buttons within a group correspond to one another.

With text-based inputs, the value of an input is determined by what a user types in; with radio buttons a user is making a multiple-choice selection. Thus, we have to define the input value. Using the `value` attribute, we can set a specific value for each `<input>` element.

Additionally, to preselect a radio button for users we can use the Boolean attribute `checked`.

Code Example:

```
<input type="radio" name="day" value="Friday" checked> Friday  
<input type="radio" name="day" value="Saturday"> Saturday  
<input type="radio" name="day" value="Sunday"> Sunday
```

Check Boxes

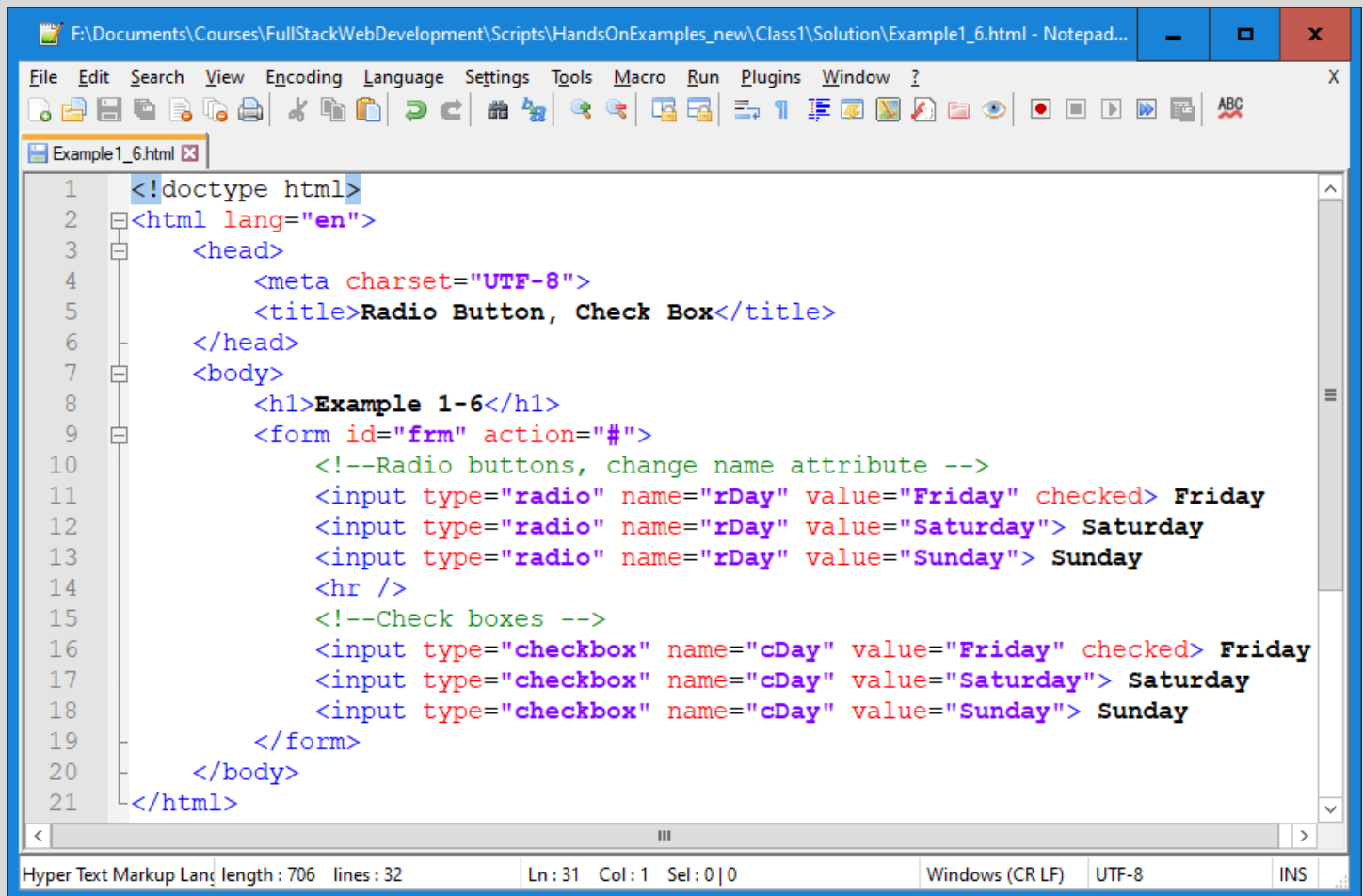
Check boxes are very similar to radio buttons. They use the same attributes and patterns, with the exception of `checkbox` as their `type` attribute value. The difference between the two is that check boxes allow users to select multiple values and tie them all to one control name, while radio buttons limit users to one value.

Code Example:

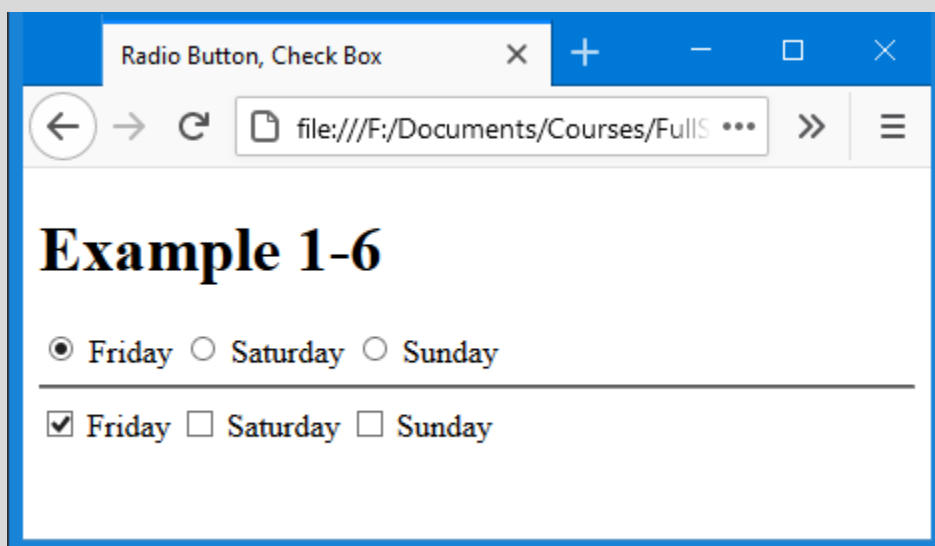
```
<input type="checkbox" name="day" value="Friday" checked> Friday  
<input type="checkbox" name="day" value="Saturday"> Saturday  
<input type="checkbox" name="day" value="Sunday"> Sunday
```


**Example 1-6:** HTML Form Example using Check Boxes and Radio Buttons

1. Create a file named Example1_6.html containing the following code:



```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Radio Button, Check Box</title>
6   </head>
7   <body>
8     <h1>Example 1-6</h1>
9     <form id="frm" action="#">
10      <!--Radio buttons, change name attribute -->
11      <input type="radio" name="rDay" value="Friday" checked> Friday
12      <input type="radio" name="rDay" value="Saturday"> Saturday
13      <input type="radio" name="rDay" value="Sunday"> Sunday
14      <hr />
15      <!--Check boxes -->
16      <input type="checkbox" name="cDay" value="Friday" checked> Friday
17      <input type="checkbox" name="cDay" value="Saturday"> Saturday
18      <input type="checkbox" name="cDay" value="Sunday"> Sunday
19    </form>
20  </body>
21 </html>
```



Radio Button, Check Box

file:///F:/Documents/Courses/FullS ...

Example 1-6

☒ Friday ☐ Saturday ☐ Sunday

☒ Friday ☐ Saturday ☐ Sunday

Drop-Down Lists/Listboxes

Drop-down lists are a perfect way to provide users with a long list of options in a practical manner. A long column of radio buttons next to a list of different options is not only visually unappealing, it is daunting and difficult for users to comprehend, especially those on a mobile device. Drop-down lists, on the other hand, provide the perfect format for a long list of choices.

To create a drop-down list we will use the `<select>` and `<option>` elements. The `<select>` element wraps all of the menu options, and each menu option is marked up using the `<option>` element.

The name attribute resides on the `<select>` element, and the value attribute resides on the `<option>` elements that are nested within the `<select>` element. The value attribute on each `<option>` element then corresponds to the name attribute on the `<select>` element.

Each `<option>` element wraps the text (which is visible to users) of an individual option within the list.

Much like the checked Boolean attribute for radio buttons and check boxes, drop-down menus can use the selected Boolean attribute to preselect an option for users.

Code Example:

```
<select name="day">
  <option value="Friday" selected>Friday</option>
  <option value="Saturday">Saturday</option>
  <option value="Sunday">Sunday</option>
</select>
```

The Boolean attribute multiple, when added to the `<select>` element for a standard drop-down list, allows a user to choose more than one option from the list at a time. Additionally, using the selected Boolean attribute on more than one `<option>` element within the menu will preselect multiple options.

The size of the `<select>` element can be controlled using CSS and should be adjusted appropriately to allow for multiple selections. It may be worthwhile to inform users that to choose multiple options they will need to hold down the Shift key while clicking to make their selections.

Code Example:

```
<select name="day" multiple>
  <option value="Friday" selected>Friday</option>
  <option value="Saturday">Saturday</option>
  <option value="Sunday">Sunday</option>
</select>
```

A listbox can be created by simply setting the size attribute to a value larger than 1.



Example 1-7: HTML Form Example using Drop-Down and Listbox

1. Create a file named Example1_7.html containing the following code:

The screenshot shows a Notepad++ editor window titled "F:\Documents\Courses\FullStackWebDevelopment\Scripts\HandsOnExamples\Class1\Solution\Example1_7.html - Notepad++". The editor contains the following HTML code:

```

1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Drop-Down, Listbox</title>
6   </head>
7   <body>
8     <h1>Example 1-7</h1>
9     <form id="frm" action="#">
10      <!--Drop-Down List -->
11      <select name="day" ><!--select Element -->
12        <option value="Friday" selected>Friday</option><!--option Element -->
13        <option value="Saturday">Saturday</option><!--option Element -->
14        <option value="Sunday">Sunday</option><!--option Element -->
15      </select><!--End select Element -->
16      <hr />
17      <!--Listbox use multiple and/or size attribute -->
18      <p>When multiple is used, use [Ctrl] + Selection to
19      select multiple rows</p>
20      <select name="day" multiple size="6"><!--select Element with multiple and size -->
21        <option value="Monday" >Monday</option><!--option Element -->
22        <option value="Tuesday" >Tuesday</option><!--option Element -->
23        <option value="Wednesday" >Wednesday</option><!--option Element -->
24        <option value="Thursday" >Thursday</option><!--option Element -->
25        <option value="Friday" selected>Friday</option><!--option Element -->
26        <option value="Saturday">Saturday</option><!--option Element -->
27        <option value="Sunday">Sunday</option><!--option Element -->
28      </select><!--End select Element -->
29    </form>
30  </body>
31 </html>

```

Below the editor, a browser window titled "Drop-Down, Listbox" is shown. The browser address bar displays "file:///F:/Documents/Courses/FullS ...". The page content includes the title "Example 1-7", a dropdown menu with "Friday" selected, and a paragraph: "When multiple is used, use [Ctrl] + Selection to select multiple rows". Below this paragraph is a listbox showing the days of the week: Tuesday, Wednesday, Thursday, Friday (highlighted), Saturday, and Sunday.

Form Buttons

After a user inputs the requested information, buttons allow the user to put that information into action. Most commonly, a submit input or submit button is used to process the data.

Submit Input

Users click the submit button to process data after filling out a form. The submit button is created using the `<input>` element with a `type` attribute value of `submit`. The `value` attribute is used to specify the text that appears within the button.

Code Example:

```
<input type="submit" name="submit" value="Send">
```

Submit Button

As an `<input>` element, the submit button is self-contained and cannot wrap any other content. If more control over the structure and design of the input is desired—along with the ability to wrap other elements—the `<button>` element may be used.

The `<button>` element performs the same way as the `<input>` element with the `type` attribute value of `submit`; however, it includes opening and closing tags, which may wrap other elements. By default, the `<button>` element acts as if it has a `type` attribute value of `submit`, so the `type` attribute and `value` may be omitted from the `<button>` element if you wish.

Rather than using the `value` attribute to control the text within the submit button, the text that appears between the opening and closing tags of the `<button>` element will appear.

Code Example:

```
<button name="submit">  
    <strong>Send Us</strong> a Message  
</button>
```

Other Inputs

Besides the applications we have just discussed, the `<input>` element has a few other use cases. These include passing hidden data and attaching files during form processing.

Hidden Input

Hidden inputs provide a way to pass data to the server without displaying it to users. Hidden inputs are typically used for tracking codes, keys, or other information that is not pertinent to the user but is helpful when processing the form. This information is not displayed on the page; however, it can be found by viewing the source code of a page. It should therefore not be used for sensitive or secure information.

To create a hidden input, you use the hidden value for the type attribute. Additionally, include the appropriate name and value attribute values.

Code Example:

```
<input type="hidden" name="tracking-code" value="abc-123">
```

File Input

To allow users to add a file to a form, much like attaching a file to an email, use the file value for the type attribute.

Code Example:

```
<input type="file" name="file">
```



Warning: Unfortunately, styling an `<input>` element that has a type attribute value of file is a tough task with CSS. Each browser has its own default input style, and none provide much control to override the default styling. JavaScript and other solutions can be employed to allow for file input, but they are slightly more difficult to construct.



Demo Example 1-4: HTML Form Example using Various HTML5 Form Elements

Execute the Demo Example

2.7 Organizing Form Elements

Knowing how to capture data with inputs is half the battle. Organizing form elements and controls in a usable manner is the other half. When interacting with forms, users need to understand what is being asked of them and how to provide the requested information.

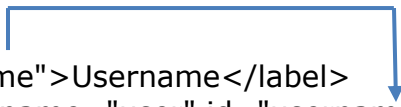
By using labels, fieldsets, and legends, we can better organize forms and guide users to properly complete them.

Label

Labels provide captions or headings for form controls, unambiguously tying them together and creating an accessible form for all users and assistive technologies. Created using the `<label>` element, labels should include text that describes the inputs or controls they pertain to.

Labels may include a `for` attribute. The value of the `for` attribute should be the same as the value of the `id` attribute on the form control the label corresponds to. Matching up the `for` and `id` attribute values ties the two elements together, allowing users to click on the `<label>` element to bring focus to the proper form control.

Code Example:



```
<label for="username">Username</label>
<input type="text" name="user" id="username">
```

If desired, the `<label>` element may wrap form controls, such as radio buttons or check boxes. Doing so allows omission of the `for` and `id` attributes.

Code Example:

```
<label>
  Username <input type="text" name="user" id="username">
</label>
```



Warning: The above wrapping of input controls inside labels might make it more difficult for styling purposes. For example, if you want to align label and input horizontally.

Fieldset

Fieldsets group form controls and labels into organized sections. Much like a `<section>` or other structural element, the `<fieldset>` is a block-level element that wraps related elements, specifically within a `<form>` element, for better organization. Fieldsets, by default, also include a border outline, which can be modified using CSS.

Code Example:

```
<fieldset>
  <label>
    Username<input type="text" name="username">
  </label>
  <label>
    Password<input type="text" name="password">
  </label>
</fieldset>
```

Legend

A legend provides a caption, or heading, for the `<fieldset>` element. The `<legend>` element wraps text describing the form controls that fall within the fieldset. The markup should include the `<legend>` element directly after the opening `<fieldset>` tag. On the page, the legend will appear within the top left part of the fieldset border.

Code Example:

```
<fieldset>
  <legend>Login</legend>
  <label>
    Username<input type="text" name="username">
  </label>
  <label>
    Password<input type="text" name="password">
  </label>
</fieldset>
```

2.8 Form & Input Attributes

To accommodate all of the different form, input, and control elements, there are a number of attributes and corresponding values. These attributes and values serve a handful of different functions, such as disabling controls and adding form validation. Described next are some of the more frequently used and helpful attributes.

Disabled

The disabled Boolean attribute turns off an element or control so that it is not available for interaction or input. Elements that are disabled will not send any value to the server for form processing.

Applying the disabled Boolean attribute to a `<fieldset>` element will disable all of the form controls within the fieldset. If the type attribute has a hidden value, the hidden Boolean attribute is ignored.

Placeholder

The placeholder HTML5 attribute provides a hint or tip within the form control of an `<input>` or `<textarea>` element that disappears once the control is clicked in or gains focus. This is used to give users further information on how the form input should be filled in, for example, the email address format to use.

The main difference between the placeholder and value attributes is that the value attribute value text stays in place when a control has focus unless a user manually deletes it. This is great for pre-populating data, such as personal information, for a user but not for providing suggestions.

Required

The required HTML5 Boolean attribute enforces that an element or form control must contain a value upon being submitted to the server. Should an element or form control not have a value, an error message will be displayed requesting that the user complete the required field.

Currently, error message styles are controlled by the browser and cannot be styled with CSS. Invalid elements and form controls (failing validation), on the other hand, can be styled using the `:optional` and `:required` CSS pseudo-classes.

Validation also occurs specific to a control's type. For example, an `<input>` element with a `type` attribute value of `email` will require not only that a value exist within the control, but also that it is a valid email address.

Additional Attributes

Other form and form control attributes include, but are not limited to, the following. Please feel free to research these attributes as necessary.

- | | | |
|-----------------------------|-------------------------------|-----------------------------------|
| ➤ <code>accept</code> | ➤ <code>formmethod</code> | ➤ <code>min</code> |
| ➤ <code>autocomplete</code> | ➤ <code>formnovalidate</code> | ➤ <code>pattern</code> |
| ➤ <code>autofocus</code> | ➤ <code>formtarget</code> | ➤ <code>readonly</code> |
| ➤ <code>formaction</code> | ➤ <code>max</code> | ➤ <code>selectionDirection</code> |
| ➤ <code>formenctype</code> | ➤ <code>maxlength</code> | ➤ <code>step</code> |



Example 1-8: Grouping Form Elements

1. Create a file named Example1_8.html containing the following code:

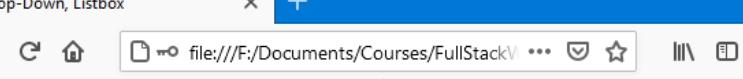
The screenshot shows a Notepad++ window titled "F:\Documents\Courses\FullStackWebDevelopment\Scripts\HandsOnExamples\Class1\Solution\Example1_8.html - Notepad...". The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, Plugins, Window, and ?. The toolbar contains various icons for file operations and editing. The active tab is "Example1_8.html".

```

1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Drop-Down, Listbox</title>
6   </head>
7   <body>
8     <h1>Example 1-8</h1>
9     <form id="frm" action="#">
10      <fieldset><!--fieldset element-->
11        <legend>Login</legend><!--legend element-->
12        <label><!--label wrapped-->
13          Username:<input type="text" name="username" required>
14        </label>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
15        <label for="password">Password</label>
16        <input type="password" name="pw" id="password">
17      </fieldset><!--End fieldset element-->
18    </form>
19  </body>
20</html>
```

The status bar at the bottom displays: Hyper Text M length : 576 lines : 31 Ln : 26 Col : 1 Pos : 567 Windows (CR LF) UTF-8 INS.

2. Execute the file in a browser:



Drop-Down, Listbox

file:///F:/Documents/Courses/FullStackV...

Example 1-8

Login

Username:

Password:



Demo Example 1-5: Advanced Form Elements

Execute the Demo Example

3. Cascading Style Sheets (CSS)

CSS is a complex language that packs quite a bit of power. It is used to format html content and add specific styles to it. In the early days, all of this formatting was done inside the HTML page. That led to confusing and spaghetti-like HTML pages. Therefore, CSS style rules are now stored in a separate file with a file extension of .css.

3.1 Overview of CSS

CSS allows us to add layout and design to our pages, and it allows us to share those styles from element to element and page to page. Before we can unlock all of its features, though, there are a few aspects of the language we must fully understand:

- First, it is crucial to know exactly how styles are rendered:
 - How do different types of selectors work?
 - How the order of those selectors can affect how our styles are rendered?
- How a few common property values that continually appear within CSS work, particularly those that deal with color and length?
- How is the cascading of various style rules processed?

Understanding Common CSS Terms

In addition to HTML terms, there are a few common CSS terms you will want to familiarize yourself with. These terms include

- selectors
- properties
- values

As with the HTML terminology, the more you work with CSS, the more these terms will become second nature.

3.2 CSS Style Rules

A CSS style rule is comprised of three main components as shown below:

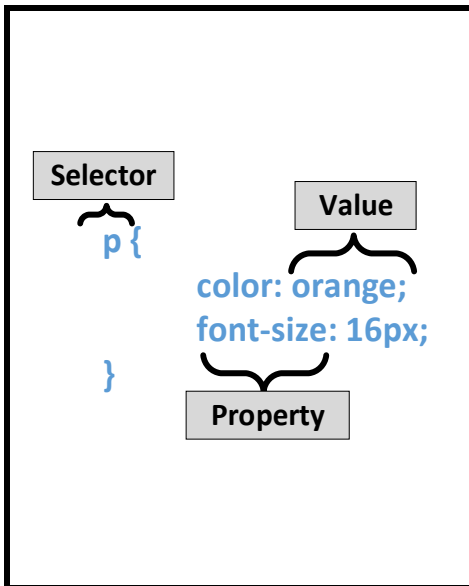


Figure 3: CSS Style Rule Components

Selectors

As elements are added to a web page, they may be styled using CSS. A selector designates exactly which element or elements within our HTML to target and apply styles (such as color, size, and position) to.

Selectors may include a combination of different qualifiers to select unique elements, all depending on how specific we wish to be. For example, we may want to select every paragraph on a page, or we may want to select only one specific paragraph on a page.

Selectors generally target an attribute value, such as an id or class value, or target the type of element, such as `<h1>` or `<p>`.

Within CSS, selectors are followed with curly brackets, `{ }`, which encompass the styles to be applied to the selected element. The selector here is targeting all `<p>` elements.

Properties

Once an element is selected, a property determines the styles that will be applied to that element. Property names come after a selector, within the curly brackets, {}, and immediately preceding a colon, :.

Code Example:

```
p {  
    color: ...;  
    font-size: ...;  
}
```

There are numerous properties we can use, such as background, color, font-size, height, and width, and new properties are often added. In the following code, we are defining the color and font-size properties to be applied to all <p> elements.

Values

So far we have selected an element with a selector and determined what style we would like to apply with a property. Now we can determine the behavior of that property with a value.

Code Example:

```
p {  
    color: orange;  
    font-size: 16px;  
}
```

Values can be identified as the text between the colon, :, and semicolon, ;. Here we are selecting all <p> elements and setting the value of the color property to be orange and the value of the font-size property to be 16 pixels.

To review, in CSS our rule set begins with the selector, which is immediately followed by curly brackets. Within these curly brackets are declarations consisting of property and value pairs. Each declaration begins with a property, which is followed by a colon, the property value, and finally a semicolon.

It is a common practice to indent property and value pairs within the curly brackets. As with HTML, these indentations help keep our code organized and legible.

3.3 Working with Selectors

Selectors, as previously mentioned, indicate which HTML elements are being styled. It is important to fully understand how to use selectors and how they can be leveraged. The first step is to become familiar with the different types of selectors. We will start with the most common selectors: type, class, and ID selectors.

Type Selectors

Type selectors target elements by their element type. For example, should we wish to target all division elements, `<div>`, we would use a type selector of `div`. The following code shows a type selector for division elements as well as the corresponding HTML it selects.

Code Example:HTML`<div>...</div>``<div>...</div>`CSS`div { ... }`

Class Selectors

Class selectors allow us to select an element based on the element's class attribute value. Class selectors are a little more specific than type selectors, as they select a particular group of elements rather than all elements of one type.

Class selectors allow us to apply the same styles to different elements at once by using the same class attribute value across multiple elements.

Within CSS, classes are denoted by a leading period, `.`, followed by the class attribute value. Here the class selector will select any element containing the class attribute value of `awesome`, including both division and paragraph elements.

Code Example:HTML`<div class="emphasize">...</div>``<p class="emphasize">...</p>`CSS`.emphasize { ... }`

ID Selectors

ID selectors are even more precise than class selectors, as they target only one unique element at a time. Just as class selectors use an element's class attribute value as the selector, ID selectors use an element's id attribute value as a selector.

Regardless of which type of element they appear on, id attribute values can only be used once per page. If used they should be reserved for significant elements.

Within CSS, ID selectors are denoted by a leading hash sign, #, followed by the id attribute value. Here the ID selector will only select the element containing the id attribute value of divMain.

Code Example:

HTML

```
<div id="divMain">...</div>
```

CSS

```
#divMain { ... }
```


Additional Selectors

Selectors are extremely powerful, and the selectors outlined here are the most common selectors we will come across. These selectors are also only the beginning. Many more advanced selectors exist and are readily available. When you feel comfortable with these selectors, do not be afraid to look into some of the more advanced selectors.

All right, everything is starting to come together. We add elements to a page inside our HTML, and we can then select those elements and apply styles to them using CSS. Now let us connect the dots between our HTML and CSS, and get these two languages working together.

3.4 Referencing CSS

In order to get our CSS talking to our HTML, we need to reference our CSS file within our HTML. The best practice for referencing our CSS is to include all of our styles in a single external style sheet, which is referenced from within the `<head>` element of our HTML document. Using a single external style sheet allows us to use the same styles across an entire website and quickly make changes sitewide.

 **Note:** Other options for referencing CSS include using internal and inline styles. You may come across these options in the wild, but they are generally frowned upon, as they make updating websites cumbersome and unwieldy.

To create our external CSS style sheet, we will want to use our text editor of choice again to create a new plain text file with a `.css` file extension. Our CSS file should be saved within the same folder, or a subfolder, where our HTML file is located.

Within the `<head>` element of the HTML document, the `<link>` element is used to define the relationship between the HTML file and the CSS file. Because we are linking to CSS, we use the `rel` (relationship) attribute with a value of `stylesheet` to specify their relationship. Furthermore, the `href` (or hyperlink reference) attribute is used to identify the location, or path, of the CSS file.

Consider the following example of an HTML document `<head>` element that references a single external style sheet.

Code Example:

```
<head>  
  <link rel="stylesheet" href="main.css">  
</head>
```


In order for the CSS to render correctly, the path of the href attribute value must directly correlate to where our CSS file is saved. In the preceding example, the main.css file is stored within the same location as the HTML file, also known as the root directory.

If our CSS file is within a subdirectory or subfolder, the href attribute value needs to correlate to this path accordingly. For example, if our main.css file were stored within a subdirectory named stylesheets, the href attribute value would be stylesheets/main.css, using a forward slash to indicate moving into a subdirectory.

At this point our pages are starting to come to life, slowly but surely. We have not delved into CSS too much, but you may have noticed that some elements have default styles we have not declared within our CSS. That is the browser imposing its own preferred CSS styles for those elements. Fortunately, we can overwrite these styles fairly easily, which is what we will do next using CSS resets.



Example 1-9: Creating, Linking and Using CSS Stylesheet in HTML Page

1. Create a file named `Example1_9.html` containing the following code:

The screenshot shows two Notepad++ editors. The top editor contains the following HTML code:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="Example1_9.css">
    <title>Using CSS Stylesheet</title>
</head>
<body>
    <h1>Example 1-9</h1>
    <div id="divForm"> <!--div element with id-->
        <form id="frm" action="#">
            <fieldset><!--fieldset-->
                <legend>Login</legend><!--legend-->
                <label><!--label wrapped-->
                    <!--Add class clsInput-->
                    Username:<input class="clsInput" type="text" name="username">
                </label>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
                <label for="password">Password:</label>
                <input type="password" name="pw" id="password">
            </fieldset>
        </form>
    </div> <!--Ending... -->
</body>
</html>
```

The bottom editor contains the following CSS code:

```
#divForm { /*ID Selector divForm */
background-color: #cacac4; /*bkg color */
height: 100px; /* height */
width: 500px; /* width */
}
input[type=password] { /*Input type password selector */
background-color: #ff9933; /*bkg color */
}
label { /*Element Selector label */
font-weight: bold; /* font weight */
}
.clsInput { /*Class Selector clsInput*/
background-color:#b96831; /*bkg color */
}
```

Login

Username:

Password:

3.5 Using CSS Resets

Every web browser has its own default styles for different elements. How Google Chrome renders headings, paragraphs, lists, and so forth may be different from how Internet Explorer does. To ensure cross-browser compatibility, CSS resets have become widely used.

CSS resets take every common HTML element with a predefined style and provide one unified style for all browsers. These resets generally involve removing any sizing, margins, paddings, or additional styles and toning these values down. Because CSS cascades from top to bottom—more on that soon—our reset needs to be at the very top of our style sheet. Doing so ensures that those styles are read first and that all of the different web browsers are working from a common baseline.

There are a bunch of different resets available to use, all of which have their own fortes. One of the most popular resets is Eric Meyer's reset (<https://meyerweb.com/eric/tools/css/reset/>), which has been adapted to include styles for the new HTML5 elements.

If you are feeling a bit more adventurous, there is also Normalize.css, created by Nicolas Gallagher (<http://necolas.github.io/normalize.css/>). Normalize.css focuses not on using a hard reset for all common elements, but instead on setting common styles for these elements. It requires a stronger understanding of CSS, as well as awareness of what you would like your styles to be.



Note: As previously mentioned, different browsers render elements in different ways. It is important to recognize the value in cross-browser compatibility and testing. Websites do not need to look exactly the same in every browser, but they should be close. Which browsers you wish to support, and to what degree, is a decision you will need to make based on what is best for your website.

In all there are a handful of things to be on the lookout for when writing CSS. The good news is that anything is possible, and with a little patience we will figure it all out.

3.6 The Cascade

We will begin breaking down exactly how styles are rendered by looking at what is known as the cascade and studying a few examples of the cascade in action. Within CSS, all styles cascade from the top of a style sheet to the bottom, allowing different styles to be added or overwritten as the style sheet progresses.

For example, say we select all paragraph elements at the top of our style sheet and set their background color to orange and their font size to 24 pixels. Then towards the bottom of our style sheet, we select all paragraph elements again and set their background color to green, as seen here.

Code Example:

```
p {  
    background: orange;  
    font-size: 24px;  
}  
p {  
    background: green;  
}
```

Because the paragraph selector that sets the background color to green comes after the paragraph selector that sets the background color to orange, it will take precedence in the cascade. All of the paragraphs will appear with a green background. The font size will remain 24 pixels because the second paragraph selector did not identify a new font size.

Cascading Properties

The cascade also works with properties inside individual selectors. Again, for example, say we select all the paragraph elements and set their background color to orange. Then directly below the orange background property and value declaration, we add another property and value declaration setting the background color to green, as shown below.

Code Example:

```
p {  
    background: orange;  
    background: green;  
}
```

Because the green background color declaration comes after the orange background color declaration, it will overrule the orange background, hence the paragraphs will appear with a green background.

All styles will cascade from the top of our style sheet to the bottom of our style sheet. There are, however, times where the cascade does not play so nicely. Those times occur when different types of selectors are used and the specificity of those selectors breaks the cascade.

3.7 Calculating Specificity


Every selector in CSS has a specificity weight. A selector's specificity weight, along with its placement in the cascade, identifies how its styles will be rendered.

We have already discussed the three different types of selectors: the type, class, and ID selectors. Each of these selectors has a different specificity weight:

- The ID selector has a high specificity weight and holds a point value of 1-0-0
- The class selector has a medium specificity weight and holds a point value of 0-1-0
- The type selector has the lowest specificity weight and holds a point value of 0-0-1

As we can see, specificity points are calculated using three columns. The first column counts ID selectors, the second column counts class selectors, and the third column counts type selectors.

What is important to note here is that the ID selector has a higher specificity weight than the class selector, and the class selector has a higher specificity weight than the type selector.

 **Note:** Specificity points are intentionally hyphenated, as their values are not computed from a base of 10. Class selectors do not hold a point value of 10, and ID selectors do not hold a point value of 100. Instead, these points should be read as 0-1-0 and 1-0-0 respectively.

The higher the specificity weight of a selector, the more superiority the selector is given when a styling conflict occurs. For example, if a paragraph element is selected using a type selector in one place and an ID selector in another, the ID selector will take precedence over the type selector regardless of where the ID selector appears in the cascade.

Code Example:HTML

```
<p id="food">...</p>
```

CSS

```
#food {  
    background: green;  
}  
p {  
    background: orange;  
}
```

Here we have a paragraph element with an id attribute value of food. Within our CSS, that paragraph is being selected by two different kinds of selectors: one type selector and one ID selector.

Although the type selector comes after the ID selector in the cascade, the ID selector takes precedence over the type selector because it has a higher specificity weight; consequently, the paragraph will appear with a green background.

The specificity weights of different types of selectors are incredibly important to remember. At times styles may not appear on elements as intended, and chances are the specificity weights of our selectors are breaking the cascade, therefore our styles are not appearing properly.

Understanding how the cascade and specificity work is a huge hurdle, and we will continue to cover this topic. For now, let us look at how to be a little more particular and intentional with our selectors by combining them. Keep in mind that as we combine selectors, we will also be changing their specificity.



Warning: CSS has an important declaration to override specificity. Use !important at the end of the selector rule but before the semicolon. The !important declaration can be easily misused if misunderstood.

3.8 Combining Selectors

So far we have looked at how to use different types of selectors individually, but we also need to know how to use these selectors together. By combining selectors we can be more specific about which element or group of elements we would like to select.

For example, say we want to select all paragraph elements that reside within an element with a class attribute value of `hotdog` and set their background color to brown. However, if one of those paragraphs happens to have the class attribute value of `mustard`, we want to set its background color to yellow. Our HTML and CSS may look like the following:

Code Example:HTML

```
<div class="hotdog">
  <p>...</p>
  <p>...</p>
  <p class="mustard">...</p>
</div>
```

CSS

```
.hotdog p {
    background: brown;
}
.hotdog p.mustard {
    background: yellow;
}
```

When selectors are combined they should be read from right to left. The selector farthest to the right, directly before the opening curly bracket, is known as the key selector. The key selector identifies exactly which element the styles will be applied to. Any selector to the left of the key selector will serve as a prequalifier.

The first combined selector above, `.hotdog p`, includes two selectors: a class and a type selector. These two selectors are separated by a single space. The key selector is a type selector targeting paragraph elements. And because this type selector is prequalified with a class selector of `hotdog`, the full combined selector will only select paragraph elements that reside within an element with a class attribute value of `hotdog`.

The second selector above, `.hotdog p.mustard`, includes three selectors: two class selectors and one type selector. The only difference between the second selector and the first selector is the addition of the class selector of `mustard` to the end of the paragraph type selector. Because the new class selector, `mustard`, falls all the way to the right of the combined selector, it is the key selector, and all of the individual selectors coming before them are now prequalifiers.

Reading the combined selector from right to left, it is targeting paragraphs with a class attribute value of `mustard` that reside within an element with the class attribute value of `hotdog`.

Different types of selectors can be combined to target any given element on a page. As we continue to write different combined selectors, we will see their powers come to life. Before we do that, though, let us take a look at how combining selectors changes a selector's specificity weight.

Specificity Within Combined Selectors

When selectors are combined, so are the specificity weights of the individual selectors. These combined specificity weights can be calculated by counting each different type of selector within a combined selector.

Looking at our combined selectors from before:

- First selector:
 - `.hotdog p`, had both a class selector and a type selector
 - Knowing that the point value of a class selector is 0-1-0 and the point value of a type selector is 0-0-1
 - The total combined point value would be 0-1-1, found by adding up each kind of selector.
- Second selector:
 - `.hotdog p.mustard`, had two class selectors and one type selector
 - Combined, the selector has a specificity point value of 0-2-1 (The 0 in the first column is for zero ID selectors, the 2 in the second column is for two class selectors, and the 1 in the last column is for one type selector)

Comparing the two selectors, the second selector, with its two classes, has a noticeably higher specificity weight and point value. As such it will take precedence within the cascade. If we were to flip the order of these selectors within our style sheet, placing the higher-weighted selector above the lower-weighted selector as shown here, the appearance of their styles would not be affected due to each selector's specificity weight.

In general, we want to always keep an eye on the specificity weights of our selectors. The higher our specificity weights rise, the more likely our cascade is to break.



Example 1-10: CSS Specificity

1. Create a file named Example1_10.html containing the following code:

```

F:\Documents\Courses\FullStackWebDevelopment\Scripts\HandsOnExamples_new\Class1\Solution\Example1_10.h...
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Example1_10.html
1 <!doctype html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <link rel="stylesheet" href="Example1_10.css">
6 <title>CSS Specificity</title>
7 </head>
8 <body>
9 <h1>Example 1-10</h1>
10 <ul id="summer-drinks"><!-- unordered list -->
11 <!--Add class favorite to any <li> item -->
12 <li>Whiskey and Ginger Ale</li> <!--li element -->
13 <li class="favorite">Wheat Beer</li><!--li element -->
14 <li>Mimosa</li><!--li element -->
15 </ul><!--End of unordered list -->
16 </body>
17 </html>
Hyper Text length : 538 lines : 28 Ln : 2

```

2. Create a file named Example1_10.css containing the following code:

CSS Specificity

file:///F:/Documents/Co

Example 1-10

- Whiskey and Ginger Ale
- **Wheat Beer**
- Mimosa

```

F:\Documents\Courses\FullStackWebDevelopment\Scripts\HandsOnExamples\Class1\Solution\...
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Example1_10.css
1 ul#summer-drinks li { /*Specificity: 1-0-2 */
2 font-weight: normal;
3 font-size: 16px;
4 color: black;
5 }
6 /*
7 1. Add !important to color and font weight
8 2. Add more specificity: ul#summer-drinks li.favorite 1-1-2
9 */
10 /* Using !important will work but not recommended
11 .favorite {
12 color: red !important;
13 font-weight: bold !important;
14 }
15 */
16 /*ul#summer-drinks li.favorite Specificity 1-1-2 */
17 ul#summer-drinks li.favorite {
18 color: red ;
19 font-weight: bold ;
20 }
length : 490 lines : 24 Ln : 24 Col : 1 Pos : 491 Windows (CR LF) UTF-8 INS

```

Layering Styles with Multiple Classes

One way to keep the specificity weights of our selectors low is to be as modular as possible, sharing similar styles from element to element. And one way to be as modular as possible is to layer on different styles by using multiple classes.

Elements within HTML can have more than one class attribute value so long as each value is space separated.

Code Example:HTML

```
<li class="important favorite standard">....</li>
```

With that, we can place certain styles on all elements of one sort while placing other styles only on specific elements of that sort.

We can tie styles we want to continually reuse to one class and layer on additional styles from another class.

Let us take a look at buttons, for example. Say we want all of our buttons to have a font size of 16 pixels, but we want the background color of our buttons to vary depending on where the buttons are used. We can create a few classes and layer them on an element as necessary to apply the desired styles.

Code Example:HTML

```
<a class="btn btn-danger">...</a>
<a class="btn btn-success">...</a>
```

CSS

```
.btn {
    font-size: 16px;
}
.btn-danger {
    background: red;
}
.btn-success {
    background: green;
}
```

Here you can see two anchor elements, both with multiple class attribute values. The first class, `btn`, is used to apply a font size of 16 pixels to each of the elements. Then, the first anchor element uses an additional class of `btn-danger` to apply a red background color while the second anchor element uses an additional class of `btn-success` to apply a green background color. Our styles here are clean and modular.

Using multiple classes, we can layer on as many styles as we wish, keeping our code lean and our specificity weights low.

**Demo Example 1-6:** Advanced CSS Stylesheet

Execute the Demo Example