Introduction to Semantic Web Development

Author: C. Sean Burns
Date: 2025-01-10
Email: sean.burns@uky.edu
Website: cseanburns.net
GitHub: @cseanburns

Introduction

This book serves as an introduction to semantic web development. A more advanced semantic web development course would introduce students to RDFS, SKOS, OWL, SPARQL or more. But this book expects undergraduate students who are new to web development. Therefore this work focuses on entry level topics: HTML5 and its semantic elements, CSS3, and JSON-LD with schema.org.

Example code and files for this work will be updated each semester I teach this course. Each semester will have its own repo. List of repos:

- Spring 2025

How to Use this Book

Markup

There are two markup styles that I want to bring to your attention:

Code Blocks

Text that looks like **code blocks** indicate some kind of command, series of commands, or web code. Do not simply copy and paste the code into your terminal or text editor. You should practice and type all code yourself.

Asides

I occasionally insert **asides** into the text. These asides generally contain notes or extra comments about the main content. Asides look like this:

> This is an aside. Asides will contain extra information, notes, or comments.

Theme

At the top of the page is an icon of a paint brush. The default theme is darker text on a light background, but you can change the theme per your preferences.

Search

Next to the paintbrush is an icon of a magnifying glass. Use this to search this work.

Printing

I intend this book to be a live document, and therefore it'll be regularly updated. But feel free to print it, if you want. You can use the print function to save the entire book as a PDF file. See the printer icon at the top right of the screen to get started.

About This Book

This book works as a live document since I use it for my spring semester Semantic Web Development course. I will update the content as I teach it in order to address changes in the technology and to edit for clarity.

This book is not a comprehensive introduction to semantic web development. It is designed for an entry level course on these topics. It is focused on a select and small range of those topics that have the specific pedagogical aims described above.

I use mdBook to build markdown source files into this final output.

Copyright

The Semantic Web

The Semantic Web is an extension of the current web. Whereas the current web was designed for people to *read*, the semantic web is designed to help machines process, intepret, and reuse data. That is, the goal of the semantic web is to allow machines (AI, etc) to understand the words on a webpage with respect to however machines understand anything. The outcome of this goal is a web of data that can be more easily processed by machines, including AI, and allow for enhanced data sharing, integration, and resuse across applications.

The vision of the semantic web originates with Tim Berners-Lee, the inventor of the web. He described the comparison between the web that we know and the semantic web as early as 2000:

> "If you think of the web today as turning all the documents in the world into one big book, then think of the Semantic Web as turning all the data into one big database ..." (Berners-Lee et al., 2000).

There are two broad ways to study the semantic web. The more advanced way is to learn how to use semantic web technologies such as:

- **Knowledge representation technologies**
  - RDF Schema (RDFS)
  - Web Ontology Language (OWL)
  - Simple Knowledge Organization System (SKOS)
- **Knowledge retrieval technologies**
  - SPARQL

However, the above technologies are more focused on underlying data models and for querying those models rather than website development. To focus on web development, in the sense that we want to create actual websites, then the semantic web technologies we are interested in are a bit different. Specifically, in this book, we will learn about:

- HTML5, which includes semantic elements.
- CSS3, which plays an important role in separating content from presentation.
- JSON-LD (JavaScript Object Notation for Linked Data), which is actually JavaScript agnostic and is used to create linked data for machines to parse and generate.
- Schema.org, which provides a collection of shared vocabularies (or terms and definitions) to annotate web content.

The Use of The Semantic Web

There are several important reasons to make data semantic on the web.

1. First, semantic data **enhanches searchability**. Recipes are a common example. When you search for a recipe, semantic data on recipe web pages allows search engines to parse the ingredients, cooking times, and other factors on those pages.
2. Second, the semantic web enables **data integration**. This is through the use of multiple vocabularies, such as Schema.org, but there are other vocabularies that can be used at the same time. This is a common use case in more advanced cases of the semantic web. For example, scientific databases can be enhanced with semantic data that integrates data from multiple scientific domains.
3. Third, the semantic web helps machines process data better and thus begin to automate complex tasks. Personal assistants like Siri and Google Assistant rely on semantic data.
4. Fourth, semantic data fosters interoperability between systems and applications. For example, a weather application can integrate data from multiple weather stations in order to provide a comprehensive forecast.

Our Goals

In this book, we will gain hands-on experience with HTML5, CSS3, and JSON-LD. These technologies will help you apply semantic data to websites and make your websites not only more searchable but also more accessible. Our tasks will include:

1. We will learn how to **structure** web pages using **HTML5** semantic elements. This will make our web content more meaningful and accessible to users and machines.
2. We will learn how to **style** and **present** our web pages using **CSS3**. Cascading Style Sheets (CSS) help us create visually appealing webpages and sites but, importantly, keep content separated from presentation.
3. We will embed structured, linked data into our webpages using **JSON-LD** and **Schema.org**. This will further enhance our content for machine-readability.
4. Finally, throughout this book, we will work on creating a final website using all of these technologies.

Conclusion

While you may not have heard of The Semantic Web, it is an important part of the evolution of the web and internet. When we make our data (content) machine-readable, we add context

and meaning to that data for machines to use. This opens up possibilities for data integration, searchability, and automation.

User Experience and Accessibility

Have you ever struggled to access a website on your phone or found a poorly designed form impossible to complete? These barriers are not just frustrating, but they are also examples of poor accessibility. Now, imagine how much more difficult and frustrating this can be when using older hardware, having a poor internet connection, or having limited eyesight. Accessible web design is considering how to create effective web pages and sites that are accessible to as many people as possible, despite the devices they have or their physical conditions.

Accessibility

In this section, we study general web accessibility and how web semantics helps make sites more accessible. I use the term accessibility as a kind of shorthand: I also include, broadly, things like usability and inclusion.

One of our readings makes important distinctions among these terms. That is, accessibility has a distinct definition, as well as the terms usability and inclusion. Our reading from w3.org defines accessibility as that which:

> addresses discriminatory aspects related to equivalent user experience for people with disabilities, including people with age-related impairments. For the web, accessibility means that people with disabilities can perceive, understand, navigate, and interact with websites and tools, and that they can contribute equally without barriers.

And then usability is related to user experience design or UX design. Usability is:

> about designing products to be effective, efficient, and satisfying. Specifically, ISO (International Organization for Standardization) defines usability as the "extent to which a product can be used by specified users to achieve specified goals effectively, efficiently and with satisfaction in a specified context of use"

Finally, inclusion is referred to as:

> Inclusive design, universal design, and design for all involves designing products, such as websites, to be usable by everyone to the greatest extent possible, without the need for adaptation. Inclusion addresses a broad range of issues including access to and quality of

hardware, software, and Internet connectivity; computer literacy and skills; economic situation; education; geographic location; and language — as well as age and disability.

Things related to inclusion are pretty broadly defined. This can be related to the responsiveness of a website to various displays, such as mobile phones, and then to the whole ranges of mobile phones that exist, as well as to desktops, laptops, and tablets, and even to smart watches.

We're going to discuss these issues. You're going to identify some websites and rank and judge these sites according to the principles of accessibility, usability, and inclusion by enabling accessibility mode on your laptop/desktop browser and on your phones.

It will help to watch a short video about accessibility on YouTube, which does a really nice job demonstrating all the above issues.

Semantics and Accessibility

Semantics refers to meaning, or what something means. Humans are pretty good (most of the time) at interpreting meaning but not computers. For example, you and I know what a person is, but a computer does not. Yet despite that, if you search for a person on the web, you will likely get accurate results back. The reason this is successful is because information retrieval algorithms use other methods to figure out to identify what is relevant based on what we search.

Thus when we talk about semantics and accessibility, we talk about how to make it so that the structure and the content on our websites can be understood by computers. The result of this is better information search and information retrieval, via search engines and artificial intelligence assistants.

It makes sense then that the meaning of a thing relates to its accessibility. Therefore, one way that we will address accessible web development is through HTML5, which was specifically released to add semantic HTML elements. This helps computer systems (like search engines and screen readers) interpret the meaning of a document's structure better than prior versions of HTML could and provide better accessibility options to people, then, too. Later we'll learn and apply JSON-LD technologies to provide meaning to the content on a page, so that a search technology can have a better understanding of what a webpage is about.

Do More with Less

There is another aspect of HTML and accessibility that we need to consider. You can think of HTML as the least common denominator among the languages that we use to develop websites. That is, a web page can be nothing more than an HTML page since it offers elements that are used to structure and provide content to a website. All other languages and technologies are additions to a web page that work and build on the HTML.

One of the goals in meeting accessibility, usability, and inclusion issues is thus to take advantage of the least common denominator aspect of HTML as much as possible. By this I mean that if you have a choice between a JavaScript (or PHP, Ruby, etc.) solution that will perform some task, and you can do the same thing in HTML, then do it in HTML. As soon as we start to make things more complicated and more sophisticated, then the more likely it becomes that we will break something, and that makes it likely to make a site less accessible, usable, and inclusive.

> While using HTML whenever possible is a best practice, there are cases where JavaScript or other languages may be required to achieve specific functionality. For example, it is necessary to use JavaScript (or another language) to create a web app. The key is to ensure that any additional complexity does not create unnecessary barriers to accessibility. Read: Google's JavaScript Warning & How it Relates to AI Search.

The Semantic Structure

I mentioned this above, but we'll talk about semantics in two ways. We we start creating our sites, we will use HTML5 to add semantics to a web page, and later we will use JSON-LD to add additional semantics that describe the content on a webpage. Some example semantic HTML5 elements include:

- `<article>` : "examples include: a forum post, a magazine or newspaper article, or a blog entry"
- `<aside>` : examples include: "sidebars or call-out boxes"
- `<figure>` : examples include: photos, plots, graphs, etc.
- `<footer>` : examples include the footer section of a website or section
- `<header>` : examples include the header section of a website (with title) or section
- `<nav>` : provides navigation links to parts of a webpage or parts of a website
- `<section>` : "represents a generic standalone section" of a webpage
- `<summary>` : "specifies a summary, caption, or legend for a `<details>` element's disclosure box"

- `<time>` : "represents a period of time

Below are links to two lists of HTML5 elements. You should begin to review them and save or bookmark these pages for constant reference:

- HTML Elements via Mozilla
- HTML Elements via W3

Attempts to provide semantic data were provided for in previous versions of HTML, like XHTML and HTML4, and through hacks that were not all that ideal, such as excessive use of the `<div>` element, or through scripting languages like JavaScript. That is, in the early days of the web, developers often used (and still do, unfortunately) non-semantic elements like `<div>` and `<span>` to structure content. In the very early days, they structured content using tables (e.g., `<table>`) elements. These were sad days because it made it difficult for assistive technologies to interpret content accurately.

However, these deficits led to the development of semantic HTML in HTML5. That is, HTML5's great benefit is that it provides semantic elements directly. For example, the `<article>` element was created to describe an article, like a blog entry, on a web page, and the `<section>` element was created to delineate a generic section on a web page. The `<nav>` element is another example of a semantic element. This element defines a site's navigation menu and helps both users and search engines quickly understand the structure of a site. Screen readers can then provide users with an overview of the site's main sections and thus improve navigation for visually impaired users.

By using such elements, we provide semantic information not just to the user of the site, or to other developers, but also to machines that parse that website for data and information. This includes web crawlers from search engines and screen readers for people who are visually impaired in some way. These semantic elements also provide text-to-speech advantages. These are a few reasons why HTML5 semantic elements are important. You won't have to memorize all of the elements, but there will be ones that you'll use more often, like the ones that I listed above.

Conclusion

In summary, web accessibility is more than a technical requirement. It is a foundational principle of inclusive design that ensures all users, regardless of ability, can access and interact with digital content. By understanding the distinctions between accessibility, usability, and inclusion, web developers can create more effective user experiences. The semantic features of HTML5 and the use of additional technologies like JSON-LD enhance accessibility by improving how content is structured and interpreted by both users and machines. Emphasizing simplicity

in web development, and prioritizing solutions that use HTML wherever possible, reduces barriers and promotes greater inclusivity. As we move forward, we should recognize the importance of semantics in web design. This will improve search and retrieval processes but also create a more equitable digital space for all users.

Tools, Setup, and Workflow

Install Software for Web Development

Web developers, and other programmers, rely on a suite of applications to conduct their work. In order to learn how to become web developers, we will need learn how to use some of these applications.

To learn to use these applications, we need to install them. Hence, our first task is to install the software we need to start web development. For this course, we will use the following tools:

- Text editor
- Vector graphics editor
- Version control software

The tricky part about this week is that I cannot show you how to install this software. The installation process is dependent on the operating system you use. Whether you use Windows, macOS, or Linux (as I do), then you should already know the basics of installing. And fortunately, once these tools are up and running on your systems, we'll all be in sync and what you'll see in my instructions going forward will be easier to follow.

Text Editor

Let's start with the most important application: a text editor. Text editors are a programmer's bread and butter. They have a long history and there are even funny cultural wars about text editors. Personally, I use the Vim editor, which is a command line editor. Vim can be difficult to learn, and we don't have the time to learn it. Therefore, for this course you can use a GUI (graphical user interface) text editor.

Note that we **do not** use a word processor like Microsoft Word or Google Docs in programming. When we write any kind of code, the code needs to be saved as plain text and not as encoded text. Word processors save files as encoded text and text editors save files as plain text (e.g., DOCX versus TXT files).

Also, text editors offer a number of functions that are designed to help us write better programs. Fortunately, many are free and open source software.

For this course, you can use the free VS Code text editor. You are welcome to use another text editor, especially if you are used to working with one and know how to use it. If you use VS

Code, feel free to explore it and see what extensions or themes you might like to add. Extensions will extend VS Code's functionality. Themes will alter how VS Code appears.

Download and install VS Code from:

- VS Code

Vector Graphics Editor

When we begin a web development project, it is a pretty bad idea to just sit down and start coding a website without first thinking about its architecture, how it looks, its design, what it contains, who the audience is or are, and so forth. The good idea is to start with a plan.

The same is true for any professional, like an architect, who designs anything in this world. Imagine having the money to build your own home and then hiring a builder who starts assembling a bunch of lumber and pipes and wires with only a vague idea of what they want to accomplish. That would be foolish as well as a waste of money and time. The same is true for any profession that builds or develops anything. If you want to build a website, then you start with a plan.

For this course, I mainly want you to think about how the website you will build will look on a desktop/laptop browser and also on mobile. We could hand draw this, but in some settings, you will want to share your plans with others, and thus it makes more sense to use a drawing tool in order to share native, digital files with your colleagues or customers.

Enter Inkscape, a vector graphics editor. Vector graphics editors are often used to design things like logos. Unlike raster-based editors, like Photoshop or GIMP, vector graphics scale to any size and still maintain quality. Adobe Illustrator is a vector graphics editor, but Inkscape is as advanced as Illustrator and is free and open source software.

If you have and are comfortable using a vector graphics editor, like Illustrator, you are free to use that, but for this course, I will use and demonstrate Inkscape.

Download and install Inkscape:

- Inkscape

Version Control

Finally, one of other most important tools for actual web development (and for any kind of programming) is version control. Version control is about project management, such as keeping track of your work and the history of your work, and about collaboration, such as

working with other developers. A number of version control software systems are available, but one of the most popular ones is Git, and we'll use that in this course.

Version control systems are enhanced with version control repositories, which are used to send, store, and share code and other work. We could set up our own Git repository, but for this course, we will use GitHub.

I'll show you the basics of using Git and GitHub in this course, but for now, we need to create an account on GitHub, if you don't have one, and download Git on our machines.

GitHub

First, create an account on GitHub, if you don't already have an account. Use a personal email address and not your university email (since it's temporary) when setting up the account:

- Create an account on GitHub

Install Git on Windows

For Windows users, I advise you follow the instructions in this video and on this page:

- Git on Windows: Do not proceed after the 1 minute and 50 second mark

Install Git on macOS

Follow the instructions in the macOS section on Installing on macOS.

Gitting Started

Git Configuration

Once Git is downloaded, we need to configure our system to use it. We do that by giving Git our name, GitHub username, and email address plus some other details. Here we need to be sure to use the same email address that we used to create our accounts on GitHub.

To get started, open a command shell or terminal on your computer (e.g., CMD.exe or PowerShell on Windows or Terminal.app on macOS) and run the following command. Note the quotes around the name but not around the **github_username** or email address. Use YOUR NAME AND YOUR EMAIL ADDRESS.

```
git config --global user.name "Your Name"
git config --global user.name github_username
git config --global user.email youremail@example.com
```

Next, we configure Git to use **main** as our default branch name. If you are using VS Code, then the second command instructs Git to use it as your default text editor. Run these two commands as-is, but if you are using a different text editor, then be sure to lookup the appropriate command for that editor:

```
git config --global init.defaultBranch main
git config --global core.editor "code --wait"
```

We can verify the above settings with the following command:

```
git config --list
```

For additional details, see the Git documentation on getting started:

- Getting Started - First-Time Git Setup

We'll soon begin to use Git and GitHub when we start coding our websites. Next, we need to understand some basic concepts with Git and GitHub.

Git Basics

Repos

The first Git concept to learn is the repository concept. Git uses two kinds of repositories:

- local repository (repo)
- remote repository (repo)

The local repo is a project directory (or folder) on your computer. I will use the term directory and not folder since the former term is more commonly used in tech fields. The project directory contains all the project files and any sub-directories for the project.

The remote repo is where we send, retrieve, or sync the files and directories that are contained in the local repo. We can retrieve projects from other repos that other people or organizations have created, if those repos are public.

With Git and GitHub, we can start a project on the local system (i.e., our computers) or start a project by creating a remote repo on GitHub and then copying it to our local system.

Branches

The second Git concept to learn is:

- branches

When you configure a directory on your local system to become a Git project, you create a default branch for your project. For small projects, we might only work in the default branch. The default branch will be named **main**.

However, since Git is a version control system, we can create additional branches to test or work on different components of our projects without messing with the main branch. For large or complex projects, we would work and switch among different branches. A large project might be a big website, an software application, or even an operating systems. Working in non-main branches (e.g., a testing branch), allows us to develop components of our project without interfering with the main branch, which might be at a stable version of our project. And then when we are ready, we can merge a testing branch with our main branch, or we can delete the testing branch if we don't want to use it.

We will primarily work with the default, main branch with our projects, but you should read the Git documentation on branches.

Important note: If we create a new repository on our local machines using Git, the default branch might be called **master**. However, if we create a new repository on GitHub, the default branch will be called **main**.

There is a long history of using terms like master and slave in various technologies, and the technology industry is beginning to come to terms with this and to use more inclusive terms. You can read more about the reasons here:

- GitHub to replace 'master' with 'main' starting next month
- Tech Confronts Its Use of the Labels 'Master' and 'Slave'

Conclusion

Going forward, we will use Inkscape to design our websites. We will use VS Code with Git and GitHub to:

1. Edit and write HTML, CSS, JSON-LD in our local repo.
2. Save the edits and new code.
3. Stage the changes so that Git tracks the new changes.
4. Commit the changes with a meaningful commit message.
5. And push the changes to the remote repo.

For future reference, here's a nice cheat sheet of Git commands. Most of these commands are to be used from the command line (Windows, macOS, or Linux), and so if we explore any command line usage of Git, these will be good to have on hand.

Designing with Inkscape

Good website design begins with a solid understanding of layout. Layouts determine how information is structured visually and guide users' interactions with your content. A well-thought-out layout ensures that a website is aesthetically pleasing and functional, and it helps users easily find what they need while enhancing their overall experience.

Fortunately, there are some common web layout designs, just like there are common building layouts or home layouts. This means you do not need to invent anything new and that there is lots to explore as you think about the kind of website you want and the kind of content you want to share. To get started, review the 11 Website Layout Examples for Every Type of Page.

Although some of the layouts at the above link might seem complicated, you can start by considering two common layout structures: Single Column Layout and Two-Column Layout.

The single-column layout can also be framed as a multiple row layout. This layout is organized by rows with a header top row, navigation second row (or shared with the header row), main content third row, and footer as the last row. Although seemingly simple, the main UK website follows a single column layout, except that it contains columns within rows.

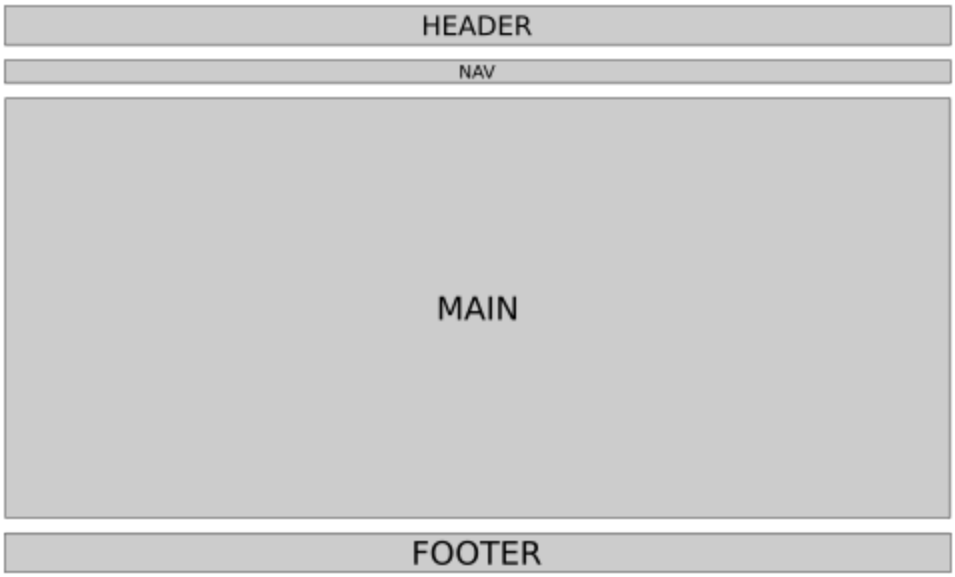


Fig. 1. Example Wireframe of a Single Column Layout

Some layouts may follow the above layout but push the navigation to the side, creating a two column layout, with the main content area to the right (usually) or left of the navigation. Many Wikipedia articles following this kind of layout.

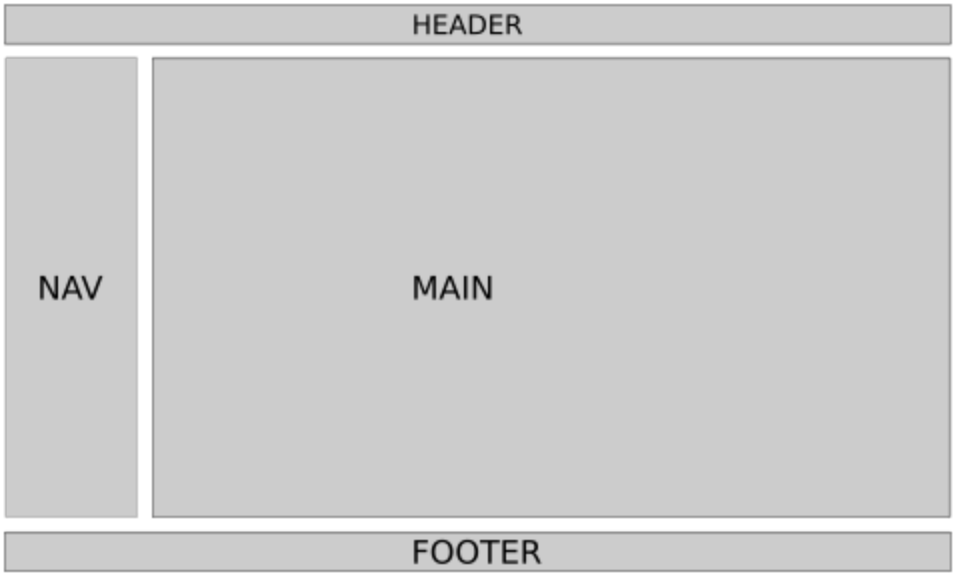


Fig. 2. Example Wireframe of a Two Column Layout

Responsive Web Design

One of the biggest challenges in modern web development is designing for multiple screen sizes. Websites must work seamlessly across devices—from large desktop monitors to small mobile screens. This is where **responsive web design** comes into play. Responsive design adapts a website's layout to screen size, resolution, and orientation of the device used to access your website.

Why Mobile Matters

We need layouts that are usable on large screens, but mobile matters for several reasons:

1. Mobile usage trends: Most Americans have a smartphone and use their smartphone browsers for accessing the web. This trend is growing around the world, too, and around 50% of global website traffic comes from mobile devices. As an example, nearly half of Wikipedia's visitors are on mobile (iOS and Android). Ignoring mobile users can therefore result in loss of engagement.
2. User expectations: Mobile users expect fast-loading, easy-to-navigate sites with content formatted for smaller screens.
3. Search engine optimization (SEO): Search engines like Google prioritize mobile-friendly websites in their rankings. This is called **mobile-first indexing**.

When we begin coding our sites using HTML and CSS, we'll have to create code that adapts to both large and small screen sizes.

Prototyping with Inkscape

Now that we've covered the importance of layout design and responsiveness, let's discuss how to prototype these designs effectively using Inkscape. Inkscape is a free, open-source vector graphics editor. It's commonly used to create illustrations (such as logos), but it's a fantastic tool for creating website layout prototypes.

Prototyping is an essential step in the web design process. It allows you to experiment with different layouts and identify issues before committing to code. With Inkscape, you can create high-quality, scalable designs for large screens and mobile views.

Getting Started with Inkscape for Layout Design

Step 1: Setting Up Your Canvas

1. Open Inkscape and create a new document.
2. Adjust the canvas size to match your target screen dimensions.
    - Go to File → Document Properties →
    - Select Custom size to pixels or **px**
    - For a regular desktop layout: Set the canvas width to 1440 pixels and height to 900 pixels (or a size relevant to your project).
    - For a mobile layout: Use dimensions like 360 pixels wide by 640 pixels tall.
    - Close Document Properties and press the number **5** to center canvas in Inskcape
3. To organize your work:
    - Save your desktop sized canvas file as **Desktop_Prototype.svg**.
    - Save your mobile sized canvas file as **Mobile_Prototype.svg**.

Step 2: Creating a Wireframe

A wireframe is a basic visual guide that represents the skeletal framework of your website. It's generally a good idea to know the content that you want to present on a site before you wireframe, but since we're in the exploration stage, we will worry about content later. When we begin to write CSS but after we have written our HTML, we can revisit and revise our wireframes as needed.

Here's how to create a wireframe in Inkscape:

1. Use the **Rectangle Tool** to define sections such as headers, footers, navigation bars, and content areas.
2. Use the **Text Tool** to label each section; e.g., "Header", "Main Content", "Footer".

3. Keep the wireframe simple by using grayscale colors to focus on layout without distractions from design details.

Step 3: Adding Design Elements

Once the wireframe is complete for your Desktop version, you can refine it by adding design elements.

1. Use shapes and gradients to visualize buttons, cards, or other interactive elements.
2. Add placeholders for images using rectangles with an **X** through them.
3. Experiment with font sizes and text alignment for headings, subheadings, and body content.
4. Save as **Desktop_Prototype.svg**.

Adapting Layouts for Mobile

To create a mobile version of your design:

1. Duplicate your desktop wireframe and adjust the canvas size to match mobile dimensions.
2. Rearrange elements to fit within the narrower viewport:
    1. Stack sections vertically
    2. Resize text and images for smaller screens
3. Ensure touch-friendly design by spacing buttons and interactive elements adequately.
4. Save as **Mobile_Prototype.svg**.

Tips for Effective Prototyping

- Keep it simple: Focus on layout and structure during prototyping. Details can be added later.
- Iterate quickly: Experiment with multiple versions to explore different layouts.
- Test early: Share prototypes with peers or users to gather feedback before coding.
- See more tips at Wireframing your layout.

Closing

Designing for both desktop and mobile views is an essential skill in web development. Using Inkscape for layout prototyping helps you visualize your designs, test ideas, and build

responsive websites effectively. In the next two chapters, we will explore how to implement these layouts using HTML first and then CSS. For now, experiment with Inkscape and create your own prototypes for both desktop and mobile views.

HTML5 and The Web

In chapter two, we focus on learning HTML5. The goal is to cover:

- Document structure metadata
- Section elements
- Grouping content and text-level semantic elements
- Links and attributes
- Embedded content: images and multimedia
- Tables and forms

In this chapter, I want you to focus solely on learning HTML5. Do not attempt to apply CSS to your content. This will mean that your web content will appear bland, but that's okay. We want to learn one thing at a time and take the time to learn that one thing.

Your Work

As we learn this material, you will do the following:

- Create files in your local repository that modify the lessons covered here.
- Push your files to GitHub.
- In GitHub, begin to build your own website using the skills you learn in this chapter.
- Remember, use HTML5 only. We'll apply CSS3 in the next chapter.

Document Structure and Metadata

Introduction

One of the main purposes of HTML is to structure our web documents and our content. This means that HTML offers a set of **elements** for this purpose. In this section, we will list, describe, and illustrate how to use these elements.

First, as you learn HTML, please refer to the official documentation at WHATWG. This is the working group for managing the HTML standards, its specifications, and documentation. The documentation for HTML5 should be constantly referenced throughout this work. See:

- HTML5 at WHATWG

Mozilla also maintains a solid reference source:

- MDN Web Docs: HTML

The Structure of an HTML Document

The overall basic structure of an HTML document is fairly simple and straightforward. An HTML document begins with the **DOCTYPE** declaration, which declares the document to be HTML. This is followed by the **HTML element** `<html>`, which "represents the root of an HTML document". The `<html>` element usually contains a **lang** attribute that we use to declare the natural language used in the document. In our case, that's English, and thus we use `<html lang="en">`. Using this helps speech synthesis tools and translation tools to parse our web document (HTML Element). The **HEAD** `<head>` section includes the document metadata and follows the HTML root element. The **BODY** `<body>` section follows and contains the document's content. The very basic structure of an HTML document looks like this:

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

You should note that most but not all HTML elements have a start or opening tag and an end or closing tag. In the above example, the opening tag `<html>` begins on the second line and the ending tag is at the last line with `</html>`. This tag therefore encompasses the `<head>` and `<body>` elements, which themselves will encompass other elements.

The <head> section

Let's focus on the HEAD section, which contains a web document's metadata. A document's metadata is data that describes the data on the page. This section of the HTML document typically includes five elements: `<title>`, `<base>`, `<link>`, `<meta>`, and `<style>`, as shown below:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title></title>
        <base>
        <link>
        <meta>
        <style></style>
    </head>
    <body>
    </body>
</html>
```

Like the `<html>` element, the `<head>` element encompasses its own elements (`<title>`, `<base>`, `<link>`, `<meta>`, and `<style>`) since it closes after them with the `</head>` element.

The <title> element

The `<title>` element contains the document's title or name. As an example, if I am writing a web page about *Linux systems administration*, then I may include that within the `<title>` element, like so:

```
<title>Using the Linux OS for Systems Administration</title>
```

Note the `<title>` element closes with `</title>`.

We can add the title information to our `<head>` section, as shown below:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using the Linux OS for Systems Administration</title>
        <base>
        <link>
        <meta>
        <style>
    </head>
    <body>
    </body>
</html>
```

The <base> element

The `<base>` element "allows authors to specify the document base URL for the purpose of relative URLs" (The base element).

The content of the `<base>` element depends on your website's domain name. If I have a website at `https://www.example.org/`, then I add this information to my `<base>` element:

```
<base href="https://www.example.org/">
```

This allows me to link to other parts of my website without using the full URL or path. For example, if I have a website with two pages: **index.html** and **news.html**, then to link to **news.html** from the **index.html** page, I only have to name the file:

```
Read my <a href="news.html">weekly newsletter</a>
```

If I didn't include the `<base>` element in my `<head>` section, then I would have to reference or link to the full URL, like so:

```
Read my <a href="https://www.example.org/news.html">weekly newsletter</a>
```

The difference between these two examples is that the first example resolves the link relative to the `<base>` URL, while the second example specifies the full domain and path. Using the `<base>` element therefore makes it easier to link to other parts of our websites. It also makes it easier if we change the domain name of our website. In such a case, we would only have to update the URL in the `<base>` element.

Let's modify our `<head>` section with this new information:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using the Linux OS for Systems Administration</title>
        <base href="https://www.example.org/">
        <link>
        <meta>
        <style>
    </head>
    <body>
    </body>
</html>
```

Note that the `<base>` element does not close.

The <link> element

The `<link>` element "allows authors to link their document to other resources" (The link element). Other resources may include external CSS stylesheets, JavaScript code, and more. We will use this element later when we link to our CSS code or stylesheets.

As an example, we might create a directory in our root project directory called **css** and include our CSS code in that directory in a file named `style.css`. In this case, we use the `<link>` element to link to our stylesheet in the following way:

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

In that example, the **rel** and the **href** are called **attributes**. The **rel** attribute describes the resource and the **href** attribute describes the location of the resource. Since we are using the `<base>` element, as described above, we only refer to the **relative** path of the stylesheet (named `style.css`) instead of the full path.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using the Linux OS for Systems Administration</title>
        <base href="https://www.example.org/">
        <link rel="stylesheet" href="css/style.css">
        <meta>
        <style>
    </head>
    <body>
    </body>
</html>
```

Note that the `<link>` element does not close.

The `<meta>` element

The `<meta>` element "represents various kinds of metadata that cannot be expressed using the" other `<head>` elements, such as `<title>`, `<base>`, `<link>`, `<style>` (The meta element). The `<meta>` element has a few use cases. To keep it simple for now, we'll cover the basics below and expand on them as we cover other topics.

First, as the documentation states, the `<meta>` element requires one of three attributes: **name**, **http-equiv**, and **charset**. We'll skip the second attribute and discuss the first and third one.

The **charset** and **name** attributes

The **charset** attribute establishes the document's character set and allows proper displaying of non-ASCII characters. In practice, we use this to declare the document's character set to UTF-8.

The **name** attribute is used with several names that refer to the web/HTML document's metadata. We'll focus on the following four:

- viewport: controls the size and shape of the browser's viewport:
  - we use this to make our site mobile friendly by default
- author: names the author of the web document:
  - we use this to name the author of the web document
- description: describes the content of the web document:
  - we use this to describe the content of the web document
- keywords: lists descriptive keywords that describe the content of the web document:
  - we use this to list keywords that describe the content of the web document

Putting this altogether, we have the following:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using the Linux OS for Systems Administration</title>
        <base href="https://www.example.org/">
        <link rel="stylesheet" href="css/style.css">
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="author" content="C. Sean Burns">
        <meta name="description" content="An introduction to Systems
Administration with Linux OS, including Debian and Ubuntu">
        <meta name="keywords" content="systems administration, linux, debian,
ubuntu">
        <style>
    </head>
    <body>
    </body>
</html>
```

The `<style>` element

Finally, the `<style>` element is used to embed CSS in our `<head>` section. Most of the time, we want to use an external stylesheet for CSS code, which we'll cover later. We want this because it means that we only need to modify one CSS file to style our entire site. That's particularly nice when our website grows to include many pages. However, there are times when we want to use a CSS style to modify just one page or some aspect of a single page.

When we use the `<style>` element, we use CSS syntax, and not HTML, within the `<style>` tag. As an example, if we want to specify a specific family of fonts, then we can use the CSS `html` **selector** to specify a font-family for a single webpage.

A CSS **selector** is often equivalently named to an HTML **element**. Such cases are called **Type selectors**. Thus, to modify the `<html>` element on a page, we use the CSS `html` type selector. In the following example, we state that the monospace font family should be used for the entire page, since we attach it to the `html` selector. On a multi-page website, this styling would only apply to this one page.

```
<style>
    html { font-family: monospace; }
</style>
```

Note that the `<style>` element requires a closing tag.

Once we add that to the `<head>` section, we have the following:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using the Linux OS for Systems Administration</title>
        <base href="https://www.example.org/">
        <link rel="stylesheet" href="css/style.css">
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="author" content="C. Sean Burns">
        <meta name="description" content="An introduction to Systems
Administration with Linux OS, including Debian and Ubuntu">
        <meta name="keywords" content="systems administration, linux, debian,
ubuntu">
        <style>
            html { font-family: monospace; }
        </style>
    </head>
    <body>
    </body>
</html>
```

Conclusion

In this section, we introduced the basic structure of a web page and covered the basics of a web document's metadata section. The latter included the following elements:

- `<!DOCTYPE html>`
- `<html>`
    - `<head>`
        - `<title>`
        - `<base>`
        - `<link>`
        - `<meta>`
        - `<style>`

In the next section, we focus on adding content within the `<body>` element of a web document.

HTML Section Elements

Introduction

In the last section, we covered the basic outline of a web document and built its metadata section in the web document's `<head>` section. At the end of that section, we created the following content:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using the Linux OS for Systems Administration</title>
        <base href="https://www.example.org/">
        <link rel="stylesheet" href="css/style.css">
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="author" content="C. Sean Burns">
        <meta name="description" content="An introduction to Systems
Administration with Linux OS, including Debian and Ubuntu">
        <meta name="keywords" content="systems administration, linux, debian,
ubuntu">
        <style>
            html { font-family: monospace; }
        </style>
    </head>
    <body>
    </body>
</html>
```

In this section, we shift our focus to the content section of our web document. This is the area within the opening and closing `<body></body>` element. We'll begin with what are called the section elements and use this to create a broad outline of our web document. In the next section, we'll cover the grouping elements. We'll use the grouping elements to begin adding content. For now, let's focus on sections.

Section Elements

There are ten main HTML section elements, or elements used to define the sections of an HTML document. We have already covered the first one, which is the `<body>` element. The `<body>` element is the only required section element. The rest of the elements help us

structure the main layout of our web document. It is not necessary to use them all, but the more structure our document has, the easier it will be for both visitors and machines to parse our web documents.

These section elements include:

1. the `<body>` element
2. the `<article>` element
3. the `<section>` element
4. the `<nav>` element
5. the `<aside>` element
6. the `<h1>`, `<h2>`, ..., `<h6>` elements
7. the `<hgroup>` element
8. the `<header>` element
9. the `<footer>` element
10. the `<address>` element

See: HTML Sections

We have already covered the `<body>` element. All remaining section elements are enclosed in the `<body>`.

The `<article>` Element

The `<article>` element contains "a complete, or self-contained, composition in a document, page, application, or site and that is, in principle, independently distributable or reusable" (The article element). Examples from WHATWG include parts of a web page like a forum post or a magazine article.

A web document may include multiple `<article>` elements in a web page, such as with forum posts, but each instance will indicate a separate, independent piece. The `<article>` element may also be nested, which means that there may exist articles within articles.

```
<body>
    <article>
    </article>
</body>
```

The <section> Element

The `<section>` element is a more generic version of the `<article>` element. It functions as "a generic section of a document or application" (The section element). The WHATWG documentation provides examples such as chapters, parts of a home page, etc. Another example might be a recipe for a dish, where the sections of a recipe include ingredients, methods, etc. We can enclose a `<section>` element within an `<article>` element or use it as a standalone element.

```
<body>
    <article>

        <section>
        </section>

    </article>
</body>
```

The <nav> Element

The `<nav>` element is used to contain navigational links to other parts of the site (The nav element). For example, if a website had a news page and an about page, then links to these pages would be placed in `<nav>` . This might be placed near the top of a web document.

```
<body>
    <nav>
    </nav>

    <article>

        <section>
        </section>

    </article>

</body>
```

The <aside> Element

The `<aside>` element functions as a sidebar and is used to represent extra but related information to the main content (The aside element). For example, imagine you have written an article on text editors. In this article, you might want to take note of a historical aspect of a specific text editor, like Vim, even though the main article isn't about Vim. This is the kind of

function that an `<aside>` fulfills. Note that I add an `<section>` element to illustrate how this can be used repeatedly.

```
<body>
    <nav>
    </nav>

    <article>

        <section>
        <aside></aside>
        </section>

        <section>
        </section>

    </article>

</body>
```

The <h1> through <h6> elements

The `<h1>` through `<h6>` elements represent section headers (The h1, h2, h3, h4, h5, and h6 elements). These are useful for marking the title and subtitles of a web document and can be used for the entire web document or within specific sections, such as within `<article>` or `<section>` elements. In the example below, I show how to use these repeatedly throughout a web document. However, it's best practice to use the `<h1>` element only once on a web page, given that it marks a top-level section.

```
<body>
    <nav>
    </nav>

    <article>
        <h1>Some title here</h1>
        <h2>Some subtitle here</h2>

        <section>
            <h2>Section title here</h2>
                <aside></aside>
            <h3>Subtitle here</h3>
        </section>

        <section>
            <h2>Section title here</h2>
        </section>

    </article>

</body>
```

The <hgroup> element

The `<hgroup>` element is used to bundle section header elements. It may include `<p>` elements for paragraphs. I demonstrate it here for completeness, but in reality, it's probably best to avoid using it. It's a confusing element and the `<header>` element, which I describe below, offers more versatility and semantics.

```
<body>
    <nav>
    </nav>

    <article>
        <hgroup>
            <h1>Some title here</h1>
            <p>A paragraph here, like an introduction.</p>
        </hgroup>

        <section>
            <h2>Section title here</h2>
                <aside></aside>
            <h3>Subtitle here</h3>
        </section>

        <section>
            <h2>Section title here</h2>
        </section>

    </article>

</body>
```

The <header> element

The `<header>` element is easily confused with the `<hgroup>` element but it has more versatility (The header element). Like the `<hgroup>` element, it's meant to be used to group section headings (e.g., the `<h1>` and etc. elements and even the `<hgroup>` element). A good use case is for the main heading area (i.e., the top of a web page), which could include a website's logo and navigational area. However, it can be used repeatedly throughout a web document, including in `<article>` or `<section>` elements. In practice, I generally use the `<header>` element and not the `<hgroup>` element.

```
<body>
    <header>
        <nav>
        </nav>
    </header>

    <article>
        <header>
            <h1>Some title here</h1>
            <h2>Some subtitle here</h2>
        </header>

        <p>A paragraph here, like an introduction.</p>

        <section>
            <h2>Section title here</h2>
                <aside></aside>
            <h3>Subtitle here</h3>
        </section>

        <section>
            <h2>Section title here</h2>
        </section>

    </article>

</body>
```

The <footer> element

The `<footer>` element "typically contains information about its section such as who wrote it, links to related documents, copyright data, and the like" (The footer element). Like the `<header>` element, it can be used repeatedly throughout a web document, such as at the end of an `<article>` or `<section>` element. It's more commonly used at the end of an entire web document, just before the closing `<body>` element. This use of course depends on the content of a web page. Therefore, use it where needed!

```
<body>
    <header>
        <nav>
        </nav>
    </header>

    <article>
        <header>
            <h1>Some title here</h1>
            <h2>Some subtitle here</h2>
        </header>

        <p>A paragraph here, like an introduction.</p>

        <section>
            <h2>Section title here</h2>
                <aside></aside>
            <h3>Subtitle here</h3>
        </section>

        <section>
            <h2>Section title here</h2>
        </section>

    </article>

    <footer>
    </footer>
</body>
```

The <address> element

The `<address>` element is used for contact information (The address element). It can be used throughout a web document. If it appears at the end of a `<section>` element, then it marks the contact information for perhaps the author of that section. If it appears in the `<footer>` element, then it marks the contact information (email addresses, mailing address, etc.) for the web site owners.

```
<body>
    <header>
        <nav>
        </nav>
    </header>

    <article>
        <header>
            <h1>Some title here</h1>
            <h2>Some subtitle here</h2>
        </header>

        <p>A paragraph here, like an introduction.</p>

        <section>
            <h2>Section title here</h2>
                <aside></aside>
            <h3>Subtitle here</h3>
        </section>

        <section>
            <h2>Section title here</h2>
        </section>

    </article>

    <footer>
        <address>
            <p>111 Main St., Lexington, KY</p>
        </address>
    </footer>
</body>
```

Quick Note on HTML Comments

Many programming and markup languages have the ability to add comments to code. This is true for HTML. When we comment HTML code, the browser will not display the contents within the commented section. Adding comments to your HTML code is helpful for two reasons:

1. You can leave yourself notes about parts of your web document.
2. You can *comment out* code that you don't want to use but might want to use later.

To comment out HTML code, use the comment tag:

```
<!-- insert comment here -->
```

The above line will be ignored by the browser.

You can also comment out multiple lines of HTML code, like so:

```
<!--
<section>
<h1>Some title here</h1>
<h2>Some subtitle here</h1>
</section>
-->
```

In that example, all lines between the comment tags will be ignored by the browser.

Putting It Together

With this section and the last, we have created the head of our web document, which contains the document's metadata, and a basic outline of our web document using the section elements listed on this page. Putting it together and using only those HTML elements covered so far, we might have something below: a starting web document about Linux systems administration. Note that I've commented out the line with the `<base>` element since I don't yet have a domain name for this content.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using the Linux OS for Systems Administration</title>
        <!-- <base href="https://www.example.org/"> -->
        <link rel="stylesheet" href="css/style.css">
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="author" content="C. Sean Burns">
        <meta name="description" content="An introduction to Systems
Administration with Linux OS, including Debian and Ubuntu">
        <meta name="keywords" content="systems administration, linux, debian,
ubuntu">
    </head>

    <body>
    <header>
        <h1>Introduction to Linux Systems Administration</h1>
        <nav>
        </nav>
    </header>

    <article>
        <header>
            <h2>The Linux Debian and Ubuntu Distributions</h2>
            <h3>Getting Started</h3>
        </header>

        <section>
            <header>
            <h2>Downloading Debian and Ubuntu</h2>
                <aside>Debian was founded in 1993. Ubuntu is a derivative of
Debian.</aside>
            <h3>Installing Debian in a Virtual Machine</h3>
            <h3>Installing Ubuntu in a Virtual Machine</h3>
            </header>
        </section>

        <section>
            <header>
            <h2>Updating Your Distributions</h2>
            </header>
        </section>
    </article>

    <article>
        <header>
            <h2>Introduction to the Bash Shell</h2>
            <h3>The Command Line</h3>
        </header>

        <section>
            <header>
```

```
            <h2>Knowing Where You Are</h2>
            </header>
        </section>
    </article>

    <footer>
        <address>
        </address>
    </footer>
    </body>
</html>
```

Validating Your HTML

When writing code or markup, it's easy to make mistakes. For example, we might leave out a required closing element or make a syntax error. To check for these kinds of mistakes, use the Nu Html Checker. You can paste your code into the text box on that page or later submit a URL when your site is live. If you receive errors or warnings, you will have to read the messages and look closely at your code to spot the error or issue.

Conclusion

As you develop your own `<head>` and `<body>` content, you will want to modify your use of the above elements based on your needs and what your web page contains. Have fun getting the skeleton of your web page/document written. Think a lot about the structure of your page. You'll want to modify that as you add content, which we'll cover next. But it's often helpful to have a starting outline, and the section elements covered here help with that.

Using VS Code and GitHub

Introduction

Developing a project means managing a project, which includes its source code and its documentation. To manage our project, we will use Git and GitHub with the Visual Studio (VS) Code editor. The process is fairly straightforward, and the steps are clearly outlined in Introduction to Git in VS Code. However, since we have already begun our project by creating a folder and an `index.html` file on our local machine, some additional setup guidance is necessary.

Install Git

Per the VS Code documentation, you need to install `git` on your computer first. To install `git`, visit the Git Downloads page and select your operating system. The instructions vary by operating system. If you are using Linux, I will assume `git` is already installed on your system or you know how to install it. If you are using Windows, the process should be fairly straightforward for standard `.exe` files. However, if you are using macOS, then you should install `brew` first.

Install Homebrew (`brew`) and `git` on macOS

To install Homebrew on macOS, visit the Homebrew page and following the instructions listed there. You will need to open your macOS Terminal.app, then copy and paste the command presented on the Homebrew page:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

And you should be good to go! Please read through the documentation to learn more. (Always read through the documentation.)

Now to use Homebrew to install `git`, run this command in your Terminal.app:

```
brew install git
```

Create GitHub Account

Once `git` is installed, visit GitHub and create an account if you don't already have one. Be sure to use your personal email and not your university or other non-permanent email, since you won't always have those accounts.

Local and Remote Repositories

Before we proceed, you should understand two main concepts:

- Local repository (repo)
- Remote repository (repo)

The project folder on your personal computer will function as your **local repo**. The content in the project folder (your files and directories) will be sent to a GitHub repository. This GitHub repository will function as your **remote repo**. The main goal is to keep these two repos in sync with each other, such that the work you do in one repo will get sent to the other repo.

Local to Remote

Most of the time you should be working on your local machine. So let's say that you edit your `index.html` file using VS Code on your personal computer. Once you have completed your editing work, you want to do the following things to sync it to your remote repo (i.e., to GitHub):

- `stage` the file: this tells `git` to take notice of this file
- `commit` the file: this tells `git` to record the changes in the repository (whether local or remote)
    - `commit` message: add a descriptive message when committing—this is helpful for version control and collaboration
- `push` the file: this updates the remote repository with the changes

Remote to Local

It's possible to edit and create files and directories on your remote repo (i.e., GitHub). I don't suggest doing this because it means using GitHub's web text editor. Although it's a decent online editor, it's not as powerful as VS Code. But if you do edit the remote repo through the browser, then you will want to sync any changes to your local repo. Like editing your local repo, the process involves staging and committing (with a commit) message any changes you've

made in your remote repo. However, you won't `push` your repo to your local repo. Instead, you'll pull up VS Code and `pull` your changes to your local repo.

Gitting Started

Now that you understand the basics of syncing repos, we'll proceed to using Git and GitHub with VS Code.

Again I'm mainly following the steps outlined at Introduction to Git in VS Code.

Now that we have `git` installed on our systems, open VS Code and access the Accounts menu. You will want to sign into GitHub in VS Code.

Since we've already created a local folder to work with, which should have our `index.html` file, we need to turn that local folder into a local repo. We do that by initializing the folder. In VS Code, see the instructions at Initialize a repository in a local folder. In short, the steps are to:

1. Open your project folder in VS Code
2. Go to the **Source Control** view and select **Initialize Repository**
3. Add a descriptive `commit` message and the press **Commit**
4. Press **Yes** to stage your commit
5. Then click **Publish Branch** and the sign-in and authenticate with GitHub.
6. Then select **Publish to GitHub public repository ...**.
    - We need this to be public for later work
7. Open on GitHub to view your uploaded repo!

Remote Sync

I highly encourage you to only work in VS Code and not edit files or create directories directly on GitHub. But if you do, then from your local machine, you will need to do a `pull` request in VS Code to sync the changes you made on GitHub to your local repo.

HTML Grouping and Text-Level Semantic Elements

Introduction

In this section, we learn about the most frequent HTML element types: grouping elements and text-level or inline semantic elements. Most of our content will be marked up using these two categories of elements.

The grouping elements we cover here include:

- the `<p>` element: paragraphs
- the `<hr>` element: horizontal rules
- the `<pre>` element: preformatted text
- the `<blockquote>` element: block quoted text
- the `<ol>` and the `<ul>` elements: ordered and unordered lists
    - the `<li>` element: items in ordered or unordered lists
- the `<dl>` element: description list
    - the `<dt>` element: description term
    - the `<dd>` element: description details
- the `<main>` element: dominant content
- the `<div>` element: generic container

For a full list of grouping elements, see: Grouping content

Some of these elements must be used together. I have indicated these as subitems in this above. For example, the `<ul>` and the `<ol>` elements are used to create *unordered* and *ordered* lists, respectively. However, either of these must include the `<li>` element to list items.

Likewise, we use the `<dl>` element to start a description or definition list. Within that, we use the `<dt>` element to specify a term in a description/definition list and the `<dd>` element to provide the description or definition.

We also cover a range of text-level or inline semantic elements. There are nearly 30 of these elements, but some are used more frequently than others. In this section, we'll cover the following text-level elements.

- the `<a>` element: hyperlinks

- the `<em>` element: emphasis
- the `<strong>` element: importance
- the `<small>` element: side comments
- the `<s>` element: mark inaccurate or irrelevant tet
- the `<cite>` element: mark titles of works
- the `<q>` element: mark quoted text
- the `<dfn>` element: mark defining term
- the `<abbr>` element: mark abbreviation or acronym
- the `<data>` element: mark machine-readable values
- the `<time>` element: mark date text
- the `<code>` element: mark computer code
- the `<var>` element: mark variable name
- the `<samp>` element: mark quoted computer output
- the `<kbd>` element: mark user input to computer
- the `<sub>` element: mark subscript
- the `<sup>` element: mark superscript
- the `<i>` element: mark alternate mood or voice in text
- the `<b>` element: mark text to draw attention to it but without suggesting extra importance
- the `<mark>` element: highlight text
- the `<span>` element: generic text marking
- the `<br>` element: add a line break

For the full list, see: Text-level semantics

A Quick Note on HTML Syntax

Closing Tags

Most tags have closing forms. An opening paragraph tag looks like this: `<p>` . It's closing version looks like this: `</p>` . The content for the paragraph is placed between the opening and closing tags.

However, some HTML tags are self-closing. Self-closing means that these tags do not have a closing syntax. These include tags like `<br>` , `<meta>` , `<hr>` . For tags like these, there is no such thing as: ~~`</br>`~~ , ~~`</meta>`~~ , ~~`</hr>`~~ , etc.

Here is where it can get confusing. In HTML5, some tags have optional closing forms and some tags require closing forms. For example, the `<p>` and `<li>` tags do not have to be closed with

`</p>` and `</li>`. But tags like `<div>` and `<span>` must be closed with `</div>` and `</span>`. Because of this mix of optional and required closing tags, it's best to close all tags that are not self-closing. Otherwise, you'll just get into the weeds about what needs closing or not.

Nesting Tags

We have to be careful how we close tags when including tags within tags. For example, if we want to include a small code snippet within a paragraph tag, then we have to close that code tag before closing the paragraph tag.

This is the correct usage:

```
<p>I use the Python interpreter as a calculator sometimes: <code>2 + 2</code>.</p>
```

In the above, the `<code></code>` tags are opened after the opening `<p>` tag and closed before the closing `</p>` tag.

This would be incorrect usage:

```
<p>I use the Python interpreter as a calculator sometimes: <code>2 + 2</p>.</code>
```

Elements in Use: Grouping Content

The HTML elements above are fairly straightforward, and the best way to learn them is to begin using them. To illustrate, in the text below, I use most of the grouping and text-level semantic elements listed above and then show the output.

```
<p>This is a paragraph.
If I add sentences to this paragraph on separate lines, the lines will be joined
when displayed in a browser.
I keep adding lines until I close the paragraph with its closing tag.
After this paragraph closes, I'll add a horizontal rule to separate this paragraph
from the next one.</p>

<hr>

<p>This is the next paragraph, but it's a short paragraph.</p>

<p>Sometimes I want to present computer code. I can do that like so:</p>

<pre>
#!/usr/bin/env bash

# list files in directory
for i in * ; do echo "${i}" ; done
</pre>

> This is a blockquote.
> We use it to quote multiple lines.
> Here's a goofy joke:
    > Why did the spider go to school? They wanted to be a web designer.

<p>Sometimes I like to create lists. Here's a list of Unix shells:</p>

<ol>
    <li>Bourne shell</li>
    <li>Bash</li>
    <li>C shell</li>
    <li>tcsh</li>
    <li>Kornshell</li>
    <li>zsh</li>
</ol>

<p>I can also present them as an unordered list:</p>

<ul>
    <li>Bourne shell</li>
    <li>Bash</li>
    <li>C shell</li>
    <li>tcsh</li>
    <li>Kornshell</li>
    <li>zsh</li>
</ul>

<p>Here's a definition:</p>

<dl>
    <dt>Linux</dt>
    <dd><q>An open-source Unix-like operating system</q> (source: Wikipedia
article on <cite>Linux</cite>)</dd>
```

```
    <dt>FreeBSD</dt>
    <dd><q>A free-software Unix-like operating system</q> (source: Wikipedia
article on <cite>FreeBSD</cite>)</dd>
</dl>
```

More Elements in Use: Text-level Semantics

We want to reserve formatting text when we use CSS, but a few HTML tags change how font is rendered. Confusingly, some tags render fonts in the same way. Let's view some of these examples.

The `<em>` and `<i>` tags

Both the `<em>` and the `<i>` tags will italicize text. The difference in use between them is literally one of semantics. If we want to place emphasis on some text, then we use the `<em>` tag. If we want to distinguish text from surrounding text in some way, then we use the `<i>` tag.

In this example, we want to place major emphasis (e.g., a warning) on the word **not** and so use the `<em>` tag.

```
<p>Do <em>not</em> eat that type of mushroom.</p>
```

Even though the presentation of the text will be the same in the browser, here we simply want to make a slight change to the text to make it stand out.

```
<p>I study <i>bibliometrics</i>.</p>
```

The `<strong>` and `<b>` tags

The same kind of difference holds true for the `<strong>` and `<b>` HTML tags. Content wrapped within these tags will be rendered as bold in the browser.

In this example, we use the `<strong>` tag to stress strong importance or urgency.

```
<p>Please submit your assignments <strong>by the due date</strong>.</p>
```

However in this example, we use the `<b>` tag as a purely visual marker:

```
<p>When you open your Canvas shell, visit the <b>Assignments</b> section to view
upcoming work.</p>
```

In all of the above cases, we will learn how to use CSS later to make the same font changes. Therefore, in choosing whether how you want to emphasize or bold text, consider the semantic differences. Also note, your choice will have an impact on how some applications, like screen readers, will present the text. For example, when using the `<strong>` rather than the `<b>` tags, a screen reader's voice will place emphasis on the content within the `<strong>` tag.

Other tags with formatting effects

Here are a few other HTML tags with special, semantic purposes:

```
<p>Be sure to use <mark>highlighting</mark> when taking notes. It helps when
reviewing for tests.</p>

<p>I have reduced prices by <s>10%</s> 20%!</p>

<p>The opposite of big is <small>small</small>.</p>

<p>In the <cite>Merchant of Venice</cite>, Shakespeare wrote: <q>All that glitters
is not gold</q>.
And in <cite>Pursuit of Happiness</cite>, Kid Cudi sang: <p>Everything that shine
ain't always gonna be gold</cite>.</p>

<p>I am a member of <abbr title="Association for Library and Information Science
Eduction">ALISE</abbr>.</p>

<p>The date is <time datetime="2025-01-25">January 25, 2025</time>.</p>

<p>In R, we assign a value to a variable named <var>x</var> like so: <code>x &lt;-
45</code>.</p>

<p>To print the output of variable <var>x</var> in R with the value of 45,
I enter the name of the variable, <kbd>x</kbd> and get back: <samp>[1] 45</samp>.
</p>

<p>Consider the vector <var>X</var><sub>1</sub>.</p>

<p>Did you solve: 42<sup>2</sup>?</p>
```

Conclusion

It's important to understand HTML grouping and text-level semantic elements because we use these elements to structure and present content effectively and semantically. Grouping elements like `<p>`, `<ul>`, `<blockquote>`, and `<main>` help organize content. Text-level semantic elements such as `<em>`, `<strong>`, `<code>`, and `<cite>` provide meaning within text. It's crucial to recognize the distinction between purely presentational tags, like `<b>` and `<i>`, and semantic elements, like `<strong>` and `<em>`. These elements affect accessibility and the way content is interpreted by search engines and other assistive technologies. Apply these elements thoughtfully. They will help create well-structured, readable, and meaningful web content that enhances both usability and accessibility.

Links and Attributes

Introduction

The ability for a web page to link to other resources is the defining characteristic of the web. According to the HTML specification, there are three types of HTML links. These include:

- links to external resources
- hyperlinks
- internal resource links

Of these three, **hyperlinks** are the most visible type of link. We use hyperlinks to navigate around the web, from one document, resource, or application to another.

Internal resource links are probably the second most visible type of link. We use these links to jump to different sections of a web page. A prime example is if a web page has a table of contents that we can use to skip to different sections of a page.

Lastly, **links to external resources** are the least visible type of link. These links are often used in the `<head>` section of a web document and they are used to "fetch and process" other resources, such as style sheets, favicons, or JavaScript.

We create links using four different HTML tags:

- the `<a>`, `<area>`, and `<form>` tags can be used as hyperlinks and as links to internal resources
- the `<link>` tag is used to link to external resources

In this section, we'll cover the `<a>` and the `<link>` tags and save the `<area>` and `<form>` tags for upcoming sections.

The <a> Element

The `<a>` tag, or anchor element, is the primary type of link in HTML. We use this tag to navigate the web or around a webpage, to download a resource, or to initiate an email or a phone call. Specifically, there are several use cases for the `<a>` tag, and these include:

Use Cases for <a>

1. External navigation
    - To link to another web page, we use the `<a>` tag along with the `href` HTML attribute.
    - Example: `<a href="https://www.example.com/">Visit Example</a>`
2. Internal page links
    - To jump to specific parts of the same page, which is useful for long pages with multiple sections (e.g., `<section>` element).
    - Example: `<a href="#section2">Go to Section 2</a>`
3. Email links
    - To open the user's default email client to send an email. This is useful for contact pages or in the `<footer>` section of a page.
    - Example: `<a href="mailto:someone@example.com">Email us</a>`
4. Telephone links
    - To provide mobile users a way to tap and call a phone number directly. This is useful for contact pages or in the `<footer>` section of a page.
    - Example: `<a href="tel:+12162452368">Call us</a>`
5. File downloads
    - To provide a link to download a file directly.
    - Example: `<a href="file.pdf" download>Download PDF</a>`

Common Attributes for <a>

Many HTML tags can be extended with the use of attributes or require attributes to function fully. In the above examples, the `href` is called an HTML attribute, and it's needed for the `<a>` tag to function as a link. The `<a>` tag can use the following attributes:

- `href` : This provides the destination URL or resource: i.e., the site or page you link to.
- `target` : This specifies where to open the linked document.
    - `_blank` opens the link in a new tab:
        - Example: `<a href="https://www.example.com" target="_blank">Visit Example</a>`
    - `_self` opens the link in the same window/tab. This is the default behavior, and you don't need to specify this unless you want to force the link to open in the same window/tab:
        - Example: `<a href="https://www.example.com" target="_self">Visit Example</a>`
- `rel` : This defines the relationship between the current document and the linked document.

    - `noopener noreferrer` is used with `target="_blank"` to disconnect the relationship between the linking document and the linked document when opening a link in a new tab.
        - Example: `<a href="https://www.example.com" target="_blank" rel="noopener noreferrer">Visit Example</a>`
    - `nofollow` instructs search engines not to follow the link. Search engines build a knowledge graph based on links, and the more links a page has, the more popular or relevant that page becomes. The `nofollow` is used to suppress that behavior in search engines (e.g., if you don't want to promote a page you're linking to).
        - Example: `<a href="https://www.example.com" target="_blank" rel="nofollow">Visit Example</a>`
- `title` : This provides additional information about the link and results in displaying a tooltip on hover.
    - Example: `<a href="https://www.example.com" title="Example.com">Visit Example</a>`
- `download` : This prompts the browser to download the linked file instead of navigating to it.
    - Example: `<a href="https://www.example.com/file.pdf" download>`

The `<link>` Element

The `<link>` tag is a self closing tag that's most often placed in the `<head>` section of a web page or HTML document. We use this tag to define relationships between our documents and external resources. Unlike the `<a>` tag, the `<link>` tag is not user-interactive. Instead, it provides the browser with additional context and resources it should load automatically when opening a web page. These resources may include CSS stylesheets, icons, fonts, and more.

Use Cases for `<link>`

1. Linking stylesheets
    - To tell the browser to fetch and apply CSS rules to your HTML document. In the following example, the rules are in a `styles.css` file in a local `css` directory.
    - Example: `<link rel="stylesheet" href="css/styles.css">`
2. Favicons
    - To instruct the browser to use an icon that represents your website in browser tabs, bookmarks, and other UI elements. Inkscape is a great application for creating favicons. The following example uses the `favicon.ico` file in a local `images` directory as the icon file.

    - Example: `<link rel="icon" href="images/favicon.ico">`
3. Preloading and prefetching resources
    - To instruct the browser to fetch resources, such as scripts, fonts, and images, early. This helps to optimize performance because it helps to load these resources before the page begins to render. The following example preloads and prefetches a JavaScript file `javascript.js` in a local `js` directory.
    - Example: `<link rel="preload" href="js/javascript.js" as="script">`
4. RSS feeds
    - To instruct browsers and feed readers that there is an RSS feed for the site or page.
    - Example: `<link rel="alternate" type="application/rss+xml" title="RSS" href="feed.xml">`

Common Attributes for `<link>`

Like the `<a>` tag, the `<link>` tab has some common attributes, some of which are shared between the two. Here are some common attributes for the `<link>` tag:

- `rel` : This specifies the relationship between the current document and the linked resource. This helps to inform the browser how to treat the linked resource.
    - Common values include: `stylesheet` , `icon` , `preload` , `prefetch` , `preconnect`
- `href` : This defines the URL of the resource being linked. Without it, you don't actually link to anything.
- `type` : This indicates the MIME type of the linked resource.
- `sizes` : This indicates the size of linked images, which is useful for favicons. This is useful for multi-resolution display. A `32x32` pixel favicon, for example, may be a good size for a laptop browser but you may need a bigger size for a tablet or smaller for a phone.
    - Example: `<link rel="icon" href="images/favicon.ico" sizes="32x32">`
- `crossorigin` : This is used to fetch resources from a different domain

As noted above, the `<link>` tag can be used to load a favicon for your website. These are the small icons that appear in your browser tab, bookmarks, and other UI elements. You can use Inkscape to create favicons for your website. When creating these icons, it's helpful to create multiple versions for different resolutions and use the `sizes` attribute to refer to them. In the example below, I also add the MIME type to help the browser recognize the purpose of these files. These icons would be stored in a directory called `images` in my project directory. Here is the example code:

```
<link rel="icon" type="image/png" sizes="16x16" href="images/favicon_16.png">
<link rel="icon" type="image/png" sizes="32x32" href="images/favicon_32.png">
<link rel="icon" type="image/png" sizes="72x72" href="images/favicon_72.png">
<link rel="icon" type="image/png" sizes="114x114" href="images/favicon_114.png">
```

Another common example is to use the `<link>` tag to load external fonts. We'll cover this in more detail when we learn about styling our sites with CSS. However, you should know that browsers use the fonts intalled on your operating system as the default. Thus, basic fonts will display a bit differently depending on your operating system: Windows, macOS, iOS, Android, Linux, etc. To mitigate this variance, many web developers use external fonts or download fonts to their web source code directory for their websites. There are several font services available for free, such as Google Fonts. Google Fonts provides a large range of fonts you can use for your websites and provides instructions on using them. For example, to use the Roboto font on your website, you can use this code in the `<head>` section of your page:

```
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?
family=Roboto:ital,wght@0,100..900;1,100..900&display=swap" rel="stylesheet">
```

And then add the following to your stylesheet or within the the `<style>` tag of your HTML document:

```
html {
    font-family: "Roboto", serif;
    font-optical-sizing: auto;
    font-weight: 300;
    font-style: normal;
    font-variation-settings: "wdth" 100;
}
```

Other Attributes

Two additional type of attributes I need to cover are the `id` and `class` attributes. These are commonly used in JavaScript and CSS. We'll cover them more when we learn about CSS, but the `id` attribute is also used to create **internal resource links**.

The `class` Attribute

The `class` attribute can be assigned to most HTML tags and is used to group multiple HTML elements together under a common name. This makes it easier to apply a shared style (CSS) or behavior (JavaScript) to the group of elements. For example, we can apply a class name to multiple HTML elements to style them similarily:

```
<main class="dark_background">
    <article class="light_background">
        <section class="dark_background">
        <aside class="light_background"></aside>
        </section>
    </article>
</main>
```

In CSS, we would refer to the class name to style classes named `dark_background` similarily and `light_background` similarily:

```
.dark_background {
    background-color: black;
    color: white;
}

.light_background {
    background-color: white;
    color: black;
}
```

The `id` Attribute

Whereas the `class` attribute is used to group different elements together, the `id` attribute is used to uniquely identify a single element on a page. This plays in handy for CSS and JavaScript, too, but because it's like applying a fingerprint to a single element, it can also be used to **internal resource links**.

Here's an example where we have multiple `<h3>` elements on a page, but we want to distinguish them from each other:

```
<h3 id="linux">Linux</h3>
<h3 id="unix">Unix</h3>
<h3 id="cpm">CP/M</h3>
```

In CSS, we referred to a `class` name with a dot selector: `.dark_background`. But for the `id` name, we refer to it with a pound selector: `#linux`, `#unix`, `#cpm`. For example, to style the above `h3` elements separately:

```
#linux {
    color: red;
}

#unix {
    color: blue;
}

#cpm{
    color: green;
}
```

One nice side effect of being able to uniquely fingerprint an element on a page with the `id` attribute is that you can use that attribute to link to it. This is **internal resource linking**. Thus, imagine a long web page with the above `<h3>` sections, we could create a table of contents that links to each section using the `id` attribute:

```
<a href=#linux">Jump to Linux section</a>
<a href=#unix">Jump to Unix section</a>
<a href=#cpm">Jump to CP/M section</a>
```

Global Attributes

Other attributes are available to HTML elements and are listed at Global Attributes. We will cover some of these global attributes as we progress through this work.

Conclusion

In summary, HTML links come in three flavors: links to external resources, hyperlinks, and internal resource links. We covered how to create these three types of links using the HTML `<a>` and `<link>` tags. Using the `<link>` tag, browsers process external resource links automatically. For example, this tag is used to bring in stylesheets, icons, or other external resources. We use the `<a>` tag to create hyperlinks, which serve as the defining characteristic of the web. Hyperlinks are used to navigate from site to site, page to page, or within a page, such as with internal resource links.

Embedded Content: Images and Multimedia

Introduction

It's no surprise that multimedia content dominates the web and thus it's no surprise that there are multiple HTML elements that provide methods for publishing multimedia content. In this lesson, we'll cover the following multimedia elements for embedding non-textual content in HTML documents. These include:

- The `<video>` element for embedding video content
- The `<track>` element for providing captioning for video content
- The `<audio>` element for embedding audio content
- The `<img>` element for embedding images
- The `<picture>` element to facilitate how image respond to different device/screen sizes
- The `<figure>` element to group media content, including images and video, with an optional `<figcaption>` element for captions
- The `<iframe>` element is used to embed other HTML pages in our main HTML documents

A couple of things to note about multimedia. First, multimedia content takes up more disk space and bandwidth than text content, and therefore, be mindful of the size of the multimedia files that you are embedding in your web pages. The bigger these files are, the longer they will take to load and to play, in the case of video or audio files.

Second, many hosting providers charge by bandwidth. Therefore, since multimedia content consumes more bandwidth, multimedia content is more costly to host. This is likely why many web developers embed third-party hosting solutions, such as YouTube, Vimeo, and others, to deliver video content. The cost is offset to the owners of those companies, which support their own products via ads and subscriptions. Although that may be a perfectly acceptable route for you to take as web developers, it's still worthy to know how to host and disseminate your own multimedia content, if you desire.

The `<video>` Element

The `<video>` element "embeds a media player" in a web page (The Video Embed Element). The element groups other elements, including the `<source>` element. The `<source>` element is used to source various video file formats for full browser support. In the following example, I

use the `<source>` element to refer to a video file in the `mp4` format and to a video file in the `webm` format. Both files are stored in a separate multimedia directory in the project folder. Both videos are otherwise equal, but providing options for both acts as a fallback mechanism in case a browser doesn't support one or the other format.

```
<video controls autoplay="false" muted="true" width="600">
    <source src="media/example.mp4" type="video/mp4">
    <source src="media/example.webm" type="video/webm">
    Your browser does not support the video tag.
</video>
```

The element accepts global attributes but also accepts attributes specific to the element. These include `controls`, `autoplay`, `width`, and more. The `controls` attribute adds functions to the embedded player, such as volume control, pause/resume playback, and more. The `autoplay` attribute instructs the browser to enable or disable autoplay upon loading. If set to `false`, autoplay is disabled (which is the humane thing to do for your site visitors!). The `width` attribute sets a default width size for the embedded player. An optional `height` attribute is also available. Other attributes are listed at the MDN Web Docs link above.

The <audio> Element

The `<audio>` element is used to embed an audio player in the HTML document (The Audio Embed Element). It uses some of the same attributes as the `<video>` element, such as `controls` to display specific playback controls in the player. In the example below, I source two versions of the audio for maximum browser support: an `mp3` file and an `ogg` file. Both files are stored in a multimedia directory in the project's main directory.

```
<audio controls>
    <source src="media/audio.mp3" type="audio/mpeg">
    <source src="media/audio.ogg" type="audio/ogg">
    Your browser does not support the audio tag.
</audio>
```

The <track> Element

The `<track>` element is used to provide subtitles, captions, and chapter titles for the `<video>` and `<audio>` elements. This element uses the WebVTT format, which is expressed as a text file that contains data, like subtitles, with time stamps to align the data with the audio or video. WebVTT files usually have `.vtt` filename extensions and can be created manually or

automatically in text transcription software. For example, Microsoft illustrates how WebVTT files can be created manually for closed caption video. Other software, like Zoom, automatically creates `.vtt` audio transcription files for cloud recordings.

In the example code below, I use the `<track>` element with the `kind` attribute to indicate that the `vtt` file is for subtitles. See the link above for other `kind` attributes to use.

```
<video controls autoplay="false" width="600">
    <source src="media/example.mp4" type="video/mp4">
    <track src="media/captions.vtt" kind="subtitles"
        srclang="en" label="English">
</video>
```

The <img> Element

The `<img>` element is used to display images, such as `png`, `jpg`, and other image files. The `alt` attribute provides a textual description of the image in case the image fails to load in the browser.

Images taken with a camera are often too large to be used on the web, and they should be resized to smaller versions in a photo editor. Once resized, we can use attributes like `width` and `height` to adjust how the image displays in the browser.

Note that the resized dimensions should correspond to the image's original dimensions, otherwise the image will be skewed. For example, my phone camera is configured to take photos with dimensions set at *4038 x 2268* in landscape mode and *2268 x 4032* in portrait mode. This results in a big file size (around 3.2 MBs), which would display slowly on slow or poor internet connections. Therefore, I might want to resize such images by a factor of six before using in a web document: `4038 / 6 = 673` and `2268 / 6 = 378`. Then I declare the sizes in the `width` and `height`

```
<img src="media/walking_bridge.jpg"
    alt="A walking bridge over a stream"
    width="673"
    height="378">
```

See the MDN Web Docs for more details on how to use the `<img>` element.

The <picture> Element

By default, the above `<img>` element should react responsively when resizing the display. This is because browsers apply a default CSS setting (`max-width: 100%`) to images, which we will discuss later. However, there may be times when it's helpful to provide different versions of an image by declaring multiple sources. The `<picture>` element enables this by serving entirely two different images, depending on the size of the display or device and not just resolutions (The Picture Element).

In the example below, when the display is equal to or smaller than 672 pixels wide, the image loads `walking_bridge_small.jpg` , which I've resized in a photo editor to have dimensions of `403 x 227 pixels` . When the display is equal to or greater than 673 pixels wide, the image loads `walking_bridge_big.jpg` , which has dimensions of `673 x 379 pixels` .

```
<picture>
    <source srcset="media/walking_bridge_small.jpg" media="(max-width: 672px)">
    <source srcset="media/walking_bridge_big.jpg" media="(min-width: 673px)">
    <img src="media/walking_bridge.jpg" alt="A walking bridge over a stream">
</picture>
```

Note the `<img>` tag is required in the `<picture>` element even though it primarily functions as a fallback image. The `<source>` tags in the `<picture>` element only provide options to the browser for which source file to load upon which condition is met.

The <figure> Element

We use the `<figure>` element to group media content, and that may include images or videos (The Figure Element). This element may also include an optional `<figcaption>` element, which provides a caption for the media. This enables the caption to be semantically grouped with the figure.

```
<figure>
    <img src="media/plot.png" alt="a plot">
    <figcaption>a scientific plot</figcaption>
</figure>
```

The <iframe> Element

The <iframe> element is used to embed a different HTML page within the current HTML page (The Inline Frame Element). We can use this element to embed a regular text-based HTML document in our main HTML page, but it's commonly used to display video from other sources, such as from YouTube, in our HTML documents.

For example, visit a YouTube video, click the **Share** button, and click the **Embed** option. This will provide <iframe> code to use in our HTML pages. Here's an example of a video from a great talk by the information scientist, Karen Jones Spärck, who helped develop the algorithms that search engines and other information retrieval systems use today to find relevant sources:

```
<iframe width="560" height="315"
    src="https://www.youtube.com/embed/5fYeKiebpuo?si=QcvAjF13AjhGkHMq"
    title="YouTube video player"
    frameborder="0"
    allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope;
picture-in-picture; web-share"
    referrerpolicy="strict-origin-when-cross-origin"
    allowfullscreen>
</iframe>
```

Conclusion

The HTML5 media elements discussed here help seamlessly integrate multimedia content in our HTML pages. We use the <video> and <audio> elements for rich media, the <track> element to provide accessibility, the <img> for images, the <picture> elements to provide responsive images, the <figure> element to group media, and the <iframe> element to embed other HTML pages.

Be aware that the <img> element is inherently responsive in most cases due to browser defaults, but the <picture> element allows serving different images entirely depending on device and screen size. Also, be sure to use the alt attribute for images to provide fallback text for screen readers and for when images fail to load.

HTML Tables and Forms

In the previous sections, we discussed several categories of HTML elements to use on our web pages. These categories relate to document structure, document metadata, grouping elements, text level semantic elements, links, attributes, and embedded content like images and multimedia. These elements are regularly used because they represent how we structure our pages and add content to them.

In this section, we learn how to create tables and forms to add to our web pages. These two aspects of HTML are less regularly used. Creating tables is about adding content, but specifically, tabular data only. Forms are used to gather user input and deliver that input to the web site owner or process it through, for example, a server side relational database server, like MariaDB, MySQL, PostgreSQL, etc.

We have a number of types of forms available to us. They include basic text entry forms, drop down lists, and more. One major caveat with forms is that they require additional programming to function. For example, forms are often used with PHP to process the input. However, learning PHP is beyond the scope of this work. We are also using GitHub Pages to render our sites, and GitHub does not provide back-end PHP or relational database support. Therefore, in this section, we mainly learn how to create a variety of HTML forms. We will learn how to style them with CSS in a future section.

The `<table>` Element

Tables are great for presenting tabular data: the kind of data that may be presented in spreadsheets or summarized in papers or articles (The `<table>` element). In this demo, I create a simple table that contains the following elements:

- `<table>` : This is the main table element. Other elements are enclosed within the `<table>` element.
- `<caption>` : The `<caption>` element is used to add captions to tables and should immediately follow the `<table>` element.
- `<thead>` : The `<thead>` element is semantic and defines table header information.
- `<tbody>` : The `<tbody>` element is semantic and defines table body information.
- `<tfoot>` : The `<tfoot>` element is semantic and defines table footer information.
- `<tr>` : The `<tr>` element defines table row.
- `<th>` : The `<th>` element defines header cell. This is often placed at the top row of a table:

    - It it often enclosed within the `<thead>` element but can be added to the `<tbody>` section for row headers.
- `<td>` : The `<td>` element defines a specific data cell in the table.

Tables are fairly straightforward but grow complex as more data is added to them. The following, simple table begins with the `<table>` tag and is followed by the `<caption>` tag. The `<caption>` tag provides a description of the table and its purpose. Following this, the table has three main sections, which include: `<thead>` , `<tbody>` , and `<tfoot>` . While HTML does not require indentation, indentation can help with code readability, and is therefore highly encouraged.

Additionally, one `<th>` element contains a `colspan` attribute, which is set to `5` , or five columns, in this example. Since the table only has five columns, this elements spans the width of the entire table. The `scope` attribute "defines the cells that the header (defined in the `<th>`) element relates to" (The `scope` attribute). It's useful for clearly demarcating the row and column names from the tabular data.

```
<table>
    <caption>Star Trek or Star Wars</caption>
    <thead>
        <tr>
            <th scope="colgroup" colspan="5">Attitude</th>
        </tr>
        <tr>
            <th scope="col">Films</th>
            <th scope="col">For</th>
            <th scope="col">Undecided</th>
            <th scope="col">Against</th>
            <th scope="col">Row Marginal</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <th scope="row">Star Trek</th>
            <td>92</td>
            <td>18</td>
            <td>90</td>
            <td>200</td>
        </tr>
        <tr>
            <th scope="row">Star Wars</th>
            <td>68</td>
            <td>22</td>
            <td>110</td>
            <td>200</td>
        </tr>
    </tbody>
    <tfoot>
        <tr>
            <th scope="row">Column Marginal</th>
            <td>160</td>
            <td>40</td>
            <td>200</td>
            <td>400</td>
        </tr>
    </tfoot>
</table>
```

The MDN Web Docs provide thorough documentation on creating basic and more advanced tables:

- Table basics
- Table advanced

Forms

Forms are used to collect user input. Visitors can use forms to enter data, make selections, and submit information to a web server, which may either store that information, process it, or transmit it across the internet.

The `<form>` Element

The `<form>` element is used to create a form. It takes at least two attributes: `action` and `method` (see additional details: The `<form>` element).

- `action` : The `action` attribute holds a URL or file path as a value that links to a script, like PHP, which processes the user input.
- `method` : The `method` attribute declares the type of HTTP request (GET and POST):
  - GET and POST are two HTTP methods to request data from web servers
  - The `GET` request forms a query string with name and value pairs. The data pairs are visible in URLs.
  - The `POST` request sends data to a server. Data is not visible in URLs, which makes this method more private by default.

Incomplete example of a basic `<form>` :

```
<form action="#" method="get">
</form>
```

The `<label>` Element

The `<label>` element is often associated with various forms and is used to add a caption in the user interface (The `<label>` element). This element takes a `for` attribute and is useful when a `<form>` has multiple `<input>` elements. Specifically, the `for` attribute connects the value of `for` to the value of the `id` attribute for the `<input>` element (see below).

Incomplete example of a basic `<form>` with the `<label>` element:

```
<form action="#" method="get">
    <label for="name">Enter your name: </label>
    <label for="email">Enter your email: </label>
</form>
```

The <input> Element

The `<input>` element defines the type of data for the input (The `<input>` element). Data types may include buttons, check boxes, dates, email addresses, and more. This element takes, among others, the following important attributes:

- The `type` attribute specifies the type of input.
- The `name` attribute is used to identify the data submitted to the server.
- The `required` attribute specifies that this field is required.
- The `id` attribute is for labeling unique input and allows for styling with CSS.
- The `for` attribute in `<label>` should match the `id` of the associated `<input>`.

Complete example of a basic `<form>` with `<label>` and `<input>` elements:

```
<form action="#" method="get">

    <label for="name">Enter your name: </label>
    <input type="text" name="name" id="name" required>

    <label for="email">Enter your email: </label>
    <input type="email" name="email" id="email" required>

    <input type="submit" value="Submit">

</form>
```

Form Variations

Create a button

Here's an example of adding a `<button>` to a form (The `<button>` element).

```
<form action="#" method="get">
    <button type="submit" name="button_like">Click if you like buttons.</button>
</form>
```

The <datalist> Element

Here's an example of creating a `<datalist>` (The `<datalist>` element). The `<datalist>` is used with the `<option>` element to offer a predefined list of options for the user to select. This

element is often used to suggest pre-defined options but allows for other data to be collected from the user.

```
<form action="#" method="get">
    <label for="myOS">What is your favorite OS?</label>
    <input list="OS" id="myOS" name="myOS">
    <datalist id="OS">
        <option value="GNU/Linux">
        <option value="Linux">
        <option value="BSD">
        <option value="Windows">
        <option value="macOS">
     </datalist>
</form>
```

The <select> Element

Like above, the `<select>` element offers a list of options, but the options are restricted and presented in a drop down box (The `<select>` element). The `<optgroup>` element groups logical options together. In the example below, I use `<optgroup>` to group two types of homes as either *traditional* or *alternative*.

```
<form action="#" method="get">
    <label for="home-structure">What kind of house would you like to have?</label>
    <select id="home-structure" name="home-structure">
        <option value="">--Please choose an option--</option>
        <optgroup label="Traditional">
            <option value="castle">Castle</option>
            <option value="mansion">Mansion</option>
            <option value="bungalo">Bungalo</option>
        </optgroup>
        <optgroup label="Alternative">
            <option value="treehouse">Tree House</option>
            <option value="tinyhome">Tiny Home</option>
            <option value="conversion_van">Conversion Van</option>
        </optgroup>
    </select>
</form>
```

The <legend> and <fieldset> Elements

The `<fieldset>` element is used to group multiple controls (The `<fieldset>` element). Adding a `<legend>` element informs the user what choice to select (The `<legend>` element). These elements are used to group inputs visually, but they also improve accessibility; specifically, they

help screen readers convey the purpose of the related inputs. The `name` attribute contains the same value for all three options since the user selects only one option.

```
<form action="#" method="get">
    <fieldset>
        <legend>Choose your favorite technology: </legend>
        <label><input type="radio" name="favtech" value="Pencil"> Pencil</label>
<br>
        <label><input type="radio" name="favtech" value="Smartphone">
Smartphone</label><br>
        <label><input type="radio" name="favtech" value="ERM"> ERM Device</label>
    </fieldset>
</form>
```

The <textarea> Element

The `<textarea>` element provides a way to accept multi-line content from a user (The `<textarea>` element). The number of lines and the width of the element are adjustable via the `rows` and `columns` attributes. In the example below, I set the number of rows to `5` and the width of the text area to `100` columns.

```
<form action="#" method="post">
    <label for="thoughts">What are you thinking about ICT nowadays?</label>
    <textarea id="thoughts" name="thoughts" rows="5" cols="100">Enter here:
</textarea>
</form>
```

Conclusion

Tables and forms are useful tools for websites. They allow developers to structure data and collect user input, respectively. They each have different purposes. We use tables to display tabular data, and we use forms to collect user input. I grouped these technologies together in a single section not because they are alike but because, unlike the rest of the HTML that we have learned, tables and forms are used sparingly.

With respect to forms, note that HTML (nor CSS) can process form data. Processing form data requires additional programming, such as PHP, which is often used with various database technologies, such as MySQL, MariaDB, PostgreSQL, etc. However, understanding how HTML forms work helps you understand the basics of frontend web application development.

In the next chapter, we begin to study CSS (Cascading Style Sheets). CSS will be used to improve the usability and visual appeal of our rather, so far, bland HTML pages.

CSS3 and Web Design

HTML provides structure and content for web pages. It defines elements like sections, headings, paragraphs, and links. HTML5 adds semantic elements, and these elements enhance the meaning and accessibility of our content. However, HTML is not responsible for presentation, layout, or style. This is the roll of CSS, or Cascading Style Sheets.

A fundamental principle of modern web development is the separate content from presentation. The idea is to use HTML to focus on structure and CSS on styling. By doing so, we create cleaner, more maintainable, and more adaptable web designs.

In this chapter, we'll explore how to use CSS3 to add visual appeal and layouts to our web pages and sites. We will cover techniques for adding color, customizing fonts, styling images and tables, and most importantly, arranging page layouts with CSS.

CSS: Getting Started

As stated by the World Wide Web Consortium (W3C), CSS, or *Cascading stylesheets*, is:

> a language for writing stylesheets, and is designed to describe the rendering of structured documents (such as HTML and XML) on a variety of media. (What is CSS?).

In short, when we write CSS, we create a stylesheet containing rules for document presentation. The presentation does not interfere with or alter the underlying semantics of the page.

CSS is very powerful. As an example, the CSS Zen Garden demonstrates how a single HTML file can be rendered differently simply by applying different CSS rules to it.

Way to Apply CSS

There are three ways to apply CSS to a web page. We can add **external CSS** stylesheets using the HTML `<link>` element in a web document's `<head>` section. In the example below, the stylesheet is named `style.css` and resides in a project's `css` folder (or directory). The name of the directory and file are arbitrary, but it's good to use descriptive names. However, the file must end with the `.css` file extension.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>External CSS</title>
        <link rel="stylesheet" href="css/style.css">
    </head>

<body>

</body>
</html>
```

As long as we add the above line to all the pages on our web site, we are able to apply the same set of CSS rules to all web pages on a web site.

Using external stylesheets has several advantages:

- **Efficiency:** Applies the same styles across multiple pages without duplicating code.
- **Reduced bandwidth:** Pages load faster since styles are cached by browsers.
- **Consistency:** Ensures uniform design across all pages on a site.
- **Easier maintenance:** Modify one CSS file to update styles for the entire site.
- **Minimizes errors and debugging:** A single CSS file minimizes potential errors and simplifies debugging.

However, we can also add CSS to a single page using two different methods: **internal CSS** and **inline CSS**. We add internal CSS by using the HTML `<style>` tag in the `<head>` section of our web document. When adding it to a single page, whatever CSS rules we add in the `<style>` element only apply to that single page. In the basic example below, I change the color of **all** `<p>` elements in a single web page to green using inline CSS.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Internal CSS</title>

        <style>
        p { color: green; }
        </style>
    </head>
<body>

<h1>Heading 1</h1>

    <p>This text would be green.</p>
    <p>So would this text.</p>
    <p>And also this text.</p>

</body>
</html>
```

We add inline CSS using the `style` HTML **attribute**. Unlike above, in the example below, I change the color of one specific `<p>` tag on a page at a time. In this case, the first line is the default color, and then I change the following paragraphs to green, red, and purple.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Inline CSS Example</title>
    </head>
<body>

<h1>Heading 1</h1>

    <p>This text is black.</p>
    <p style="color:green;">But this text is green.</p>
    <p style="color:red;">However, this text is red.</p>
    <p style="color:purple;">Now we're out of control: this text is purple.</p>

</body>
</html>
```

In short, using external, internal, and inline CSS is a matter of applying CSS with greater page and element specificity. External stylesheets can be used by all pages on a website. Internal CSS is used by a single page. Inline CSS is used by a single element in a single page.

Be aware that inline styles override internal and external styles and internal styles override external styles. That is, if I declare in an external stylesheet that all `<p>` elements should be red, but then use `<p style=color:green;">` on some page in my site, green will override the red for that particular case.

Conclusion

In this chapter on CSS, we will primarily use external stylesheets because they offer more advantages than internal or inline styles. External stylesheets don't take away any precision, either. When desired, there are certain methods we can use to apply CSS to single pages or to specific elements, even when CSS code is in an external stylesheet.

In the next two sections, we first learn about CSS **selectors**, which is the most important and basic aspect of CSS to know. Then will learn how to apply colors and define units of measurement. Later we will learn CSS that will allow us change fonts and add other effects. Then we will acquire an understanding of how CSS controls the elements on a page and use that understanding to create various layouts.

Syntax and Selectors

In HTML, **elements** define the structure of a webpage. Common elements include `<h1>`, `<main>`, `<p>`, `<table>`, and many others. We also learned that these elements may include **attributes**. For example, we can add global attributes, like the `class` or `id` attributes, to all tags. Furthermore, we learned that some elements, like the `<th>` tag, may have element specific attributes, like `colspan`. In order to style the HTML, CSS provides a mechanism to focus on specific elements or attributes. In CSS, this mechanism is called **selectors**.

There are multiple types of selectors. These include:

- Universal selector
- Type selectors
- Attribute selectors
- Class selectors
- ID selectors
- Pseudo classes
- Pseudo elements
- Combinators

Each of these selectors are discussed below along with examples. For a deeper understanding of each selector type, refer to the linked resources above.

Also, remember that to use these selectors, we add them to an external stylesheet for the reasons listed in the prior section. In my previous HTML discussions, I added the following line to the `<head>` section of my HTML document that links to my external stylesheet:

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

Therefore, when I begin CSS coding, I create rules using these selectors by adding them to a file called `style.css` in a `css` directory in my project repository.

CSS Syntax

CSS generally follows a very simple syntax. It begins with a selector, which for **type selectors** is equivalent to the HTML element. This is followed by an opening curly brace `{` and a list of **properties** and **values** to apply to the element named by the selector. Each property / value pair is closed with a semicolon `;`. And the selector rule ends with a closing curly brace `}`. This is expressed in the following generic example:

```
selector {
    property: value;
    property: value;
    property: value;
}
```

As a real example, if we want to change the foreground and background color of all `<p>` elements in a web document red and black, respectively, we would write the following rule:

```
p {
    color: red;
    background-color: black;
}
```

Universal Selector

The **universal selector** is indicated by a single asterisk `*` and matches all elements in a web document. In the following example, I use this selector to make all text bold:

```
* {
    font-weight: bold;
}
```

This selector should be used sparingly, but it has special and advanced use cases when combined with other selector types.

Type selectors

The **type selector** is the most important selector. The names for these selectors specifically match the names of their HTML element equivalents. As an example, the CSS `p` selector matches the HTML `<p>` element. The CSS `h1` selector matches the HTML `<h1>` element. And so forth.

In the following example, I use three type selectors: `html`, `h1`, and `p`. I use the `html` selector to change the foreground and background color to the entire HTML page, since an entire HTML page is enclose within the `<html>` element. I then show how to apply a sans-serif font-family to the `<h1>` elements, and a serif font to the `<p>` elements in a web document:

```
html {
    background-color: red;
    color: white;
}

h1 {
    font-family: sans-serif;
}

p {
    font-family: serif;
}
```

Attribute selectors

The **attribute selector** is used to style HTML elements with specific attributes. For example, if I use the `id` attribute throughout a web document, then I can apply CSS to all elements that have an `id` attribute, regardless of the element type or the value of the attribute. As an example, let's say I have a web document with the following:

```
<h1 id="test1">This is test 1.</h1>
<h2 id="test2">This is test 2.</h2>
<section id="test3">
    <p>This is test 3.</p>
</section>
```

Then the following CSS will bold the font for all three heading elements:

```
[id] {
    font-weight: bold;
}
```

We can also select attributes with specific values:

- `[att=value]` : select by attribute with specific value:
  - For example, `[id="test1"]` would select `<h1 id="test1">`
- `[att~=value]` : select by attribute where value is a separate word:
  - For example: `[id~="test"]` would match `<h1 id="test one">` and `<h2 id="test two">` since `test` is separate in each.

Class selectors

The **class selectors** focus on styling attributes with specific class names in an HTML document. Consider the following HTML that has three `<p>` tags with three separate classes: `bold`, `italics`, and `green`:

```
<p class="bold">This text is bold.</p>
<p class="italics">This text is italicized.</p>
<p class="green">This text is green.</p>
```

To use the class selector, I identify the class name in my CSS with the *dot name* `.class_name` syntax:

```
.bold {
    font-weight: bold;
}

.italics {
    font-style: italic;
}

.green {
    color: green;
}
```

ID selectors

Like the class selector, **ID selectors** identify the ID as the property and its value. To apply an ID selector, we use the *pound name* syntax `#id_name`.

Recall that while class attributes can be re-used throughout a web document, ID attributes are unique and can only be used once. Consider the following HTML:

```
<section id="news">News</section>
<section id="about">About Us</section>
<section id="store">Store</section>
```

In the above HTML snippet, there can only be one `id` with a value of `news`, one `id` with a value of `about`, and one `id` with a value of `store`. To apply CSS to those specific IDs, we select by the value of the ID attributes. In the example CSS below, I add a black, dotted border that is one pixel thick to the News section, I change the line-height to 1.5 times the element's font-size in the About section, and I convert text in the Store section to all upper case:

```
#news {
    border-bottom: 1px solid black;
}

#about {
    line-height: 1.5;
}

#store {
   text-transform: uppercase;
}
```

Combinators

Combinators, and the **pseudo classes** and **pseudo elements** discussed in the following section, should be used when you start to gain a grasp of the relationships between HTML elements on a web page. Specifically, all elements in an HTML document can be represented in a hierarchical document tree. The hierarchy of elements includes descendant elements, child elements, and sibling elements. You can view the document tree of the HTML in your web browser by accessing Web Developer Tools for your browser.

Descendant combinators

The **descendant combinator** selects "an element that is the descendant of another element in the document tree" (Descendant combinator). These are elements that are contained within other elements. For example, in the following code snippet, the `<var>` element is a descendant of the `<p>` element because it is enclosed within the `<p>` element:

```
<p>Consider the vector <var>X</var>.</p>
```

And in the following snippet, the `<h2>` element is a descendant of the `<section>` element.

```
<section>
    <h2>Heading</h2>
</section>
```

To use the descendant combinator, we simply identify both elements in our CSS. For example, to select the `<var>` element within the `<p>` element, we use the following syntax:

```
p var {
    font-size: 10pt;
}
```

In the above case, the `<var>` element is a direct descendant of the `<p>` element. However, we may also select a grandchild or later element by using the asterisk. For example, the following selects all `<p>` elements within all `<section>` elements in an HTML page:

```
section * p {
    font-size: 14pt;
}
```

Child combinators

"A **Child combinator** describes a childhood relationship between two elements" (Child combinators). A prime example of a child combinator is the HTML list. Consider the following unordered list in HTML, where the `<li>` element is a child of the `<ul>` element:

```
<ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Grapes</li>
    <li>Oranges</li>
</ul>
```

To select the `<li>` child element, we use the following CSS syntax, where I convert all items in the unordered list to lowercase.

```
ul > li {
    text-transform: lowercase;
}
```

Note that a child element is also a descendant element, so the descendant syntax (e.g., `ul li`) would also work. The child combinator is simply a more specific type of descendant.

Sibling combinators

Per the W3 documenation, "there are two different sibling combinators: the next-sibling combinator and subsequent-sibling combinator" (Sibling combinators).

The **next-sibling combinator** selects elements that immediately follow another element. Consider the following HTML, where a `<p>` element is followed by an `<hr>` (horizontal rule)

element, which itself is followed by another `<p>` element:

```
<p>42 is the answer to life, the universe, and everything in it.</p>
<hr>
<p>What is the question?</p>
```

In the above HTML snippet, the `<hr>` element is a next-sibling of the first `<p>` element, and the second `<p>` element is a next-sibling of the `<hr>` element. Therefore, to identify the last `<p>` element in the above snippet, which is the next-sibling of the `<hr>` element, we use the following next-sibling combinator:

```
hr + p {
    font-size: 20pt;
}
```

A **subsequent-sibling combinator** simply extends this relationship down the document tree. Consider the following HTML snippet, which includes a bit of the opening to Leo Tolstoy's novel *Anna Karenina*.

```
<h1>Anna Karenina / by Leo Tolstoy</h1>
<h2>Chapter 1</h2>
<p>Happy families are all alike; every unhappy family is unhappy in its own way.
</p>
<p>Everything was in confusion in the Oblonksy's house....</p>
<h2>Chapter 2</h2>
```

We could specifically identify the `<h2>` elements as subsequent-siblings of the `<h1>` element with the following CSS syntax:

```
h1 ~ h2 {
    color: yellow;
    background-color: black;
}
```

Pseudo-classes and pseudo-elements

Pseudo-classes are used to select a specific state of an element. There are several types of pseudo-classes, but we will focus on just a few from two categories. For more details on other categories of pseudo-classes as well as a full list, see Pseudo-classes.

- Dynamic pseudo-classes:
    - `:link`

    - `:visited`
    - `:hover`
    - `:active`
    - `:focus`
- Structural pseudo-classes:
    - `:nth-child()`
    - `:nth-last-child()`
    - `:first-child`

The dynamic pseudo-classes `:link` and `:visited` are often used to style hyperlinks (aka, the `<a>` anchor tag). In such situations, `:link` must be used before `:visited`. Here is an example of styling hyperlinks in CSS:

```
a:link {
    text-decoration: green wavy underline;
}

a:visited {
    text-decoration: underline overline;
}
```

The `:hover` pseudo-class is also often paired with the `<a>` anchor tag, but it can be used on other elements, too. For example, it's often useful to apply to rows in a `<table>` to help highlight rows of interest. This will trigger a style change when the mouse cursor hovers over the element:

```
a:hover {
    color: purple;
    font-size: 20pt;
}
```

Pseudo-elements are methods to "create abstractions about the document tree beyond those specified by the document language" (Psuedo-elements). For example, there is no such element to indicate the first line in a `<p>` element, but using the **pseudo-class** for `first-line`, we can focus on and style the first line of a `<p>` (or other) element. For example, consider the following HTML:

```
<p>To be, or not to be, that is the question:<br>
Whether 'tis nobler in the mind to suffer<br>
The slings and arrows of outrageous fortune,<br>
Or to take arms against a sea of troubles<br>
And by opposing end them. To die&mdash;to sleep,<br>
No more;</p>
```

Since there is a `<br>` tag at the end of each line, the first line is “To be, or not to be, that is the question:”. Using the **pseudo-element** for `first-line`, we can style it by, for example, converting it to uppercase:

```
p::first-line {
    text-transform: uppercase;
}
```

Other **pseudo-classes** that can be used include:

- `::first-letter`
- `::before`
- `::after`

Conclusion

CSS Selectors and syntax form the foundation of web styling. They allow us to precisely target and modify elements within an HTML document. When we understand **type**, **class**, **ID**, and **attribute** selectors, and then more advanced **combinators** and **pseudo-classes/elements**, we are able to create flexible, maintainable styles that enhance the design and usability of our web pages.

As you explore and apply CSS to your websites, experiment with different selectors to see how they interact. This is especially important when you work with the various combinator selectors, since these depend on how HTML elements are related to each other. It is important to become familiar with the selectors and syntax discussed here, because they are important building blocks to more advanced topics like responsive design. And although we will not use any CSS frameworks in this course, understanding the essentials here will help with those if you eventually use them.

Colors and Units of Measurements

Introduction

Designing a web page begins by controlling color and sizing. These base conditions lay the foundations for shaping user experience. It's also where the fun begins.

Importantly, colors establish a site's visual identity, improves readability, and creates contrast, which is important for website accessibility. There are multiple methods to define colors in CSS. These methods include named colors (red, blue, green, yellow, etc), RGB values, hexadecimal codes, and HSL (Hue-Saturated-Lightness) values. Each method has its strengths with respect to levels of precision and to device type.

CSS also provides many different types of units of measurement. These units allow for precise control over layout and typography, which we will cover soon, and define dimensions such as font size, spacing, and layout structure. There absolute units, such as pixels and inches, and there are relative units, such as `rem` and `em`. The latter make designs more flexible and responsive.

In this section, we will explore how to use CSS color properties to enhance design and measurement units to create scalable, adaptable layouts. These concepts will help you build visually appealing and responsible web pages.

Colors

There are four methods to create or add color to the various parts of our websites. We can use:

- Color keywords:
  - Basic color keywords: black, silver, gray, red, green, teal, etc.
  - Extended color keywords: aqua, chocolate, cyan, gold, indigo, lime, etc
- RGB values:
  - The RGB value can declare a specific color: `p { color: rgb(255, 241, 0); }` (a yellow color)
  - We can add a fourth value to indicate opacity; here I set opacity to 50%: `p { color: rgb(255, 241, 0, 0.5); }`
- Hexadecimal values;
  - We can use a hexadecimal value to identify a color: `p { color: #fffb00; }`

        - Lots of websites list RGB and Hexadecimal values, such as: RGB / Hexademical Color Picker
- HSL (hue-saturated-lightness) values:
    - HSL is considered more useful than RGB because it's less hardware dependent and considered more intuitive.
    - The values stand for Hue, Saturation, and Lightness: `p { color: hsl(291, 100%, 50%); }`
    - The Hue is a degree on a color wheel and accepts values from 0 to 360.
    - Saturation and Lightness are coded as percentages.
        - See: HSL Color Picker

We can add color to any element, and we can specify which elements to style using the selectors we reviewed in the prior section. As an example, consider the following, abbreviated HTML document:

```
<html>
    <body>
    <main>
    <h1>Heading 1</h1>
        <section>
        <h2>Heading 2a</h2>
        <p>Paragraph 1</p>
        <p>Paragraph 2</p>
        <pre><code>
        #!/usr/bin/env bash

        for i in * ; do echo "${i}" ; done
        </code></pre>
        <h2>Heading 2b</h2>
        <p>Paragraph 3</p>
        </section>
    </main>
    </body>
</html>
```

We can add background color to the page. In the code below, `aliceblue` is a color keyword and `#000000` is hexadecimal for black:

```
html {
    background-color: aliceblue;
    color: #000000;
}
```

We can colorize the headings:

```
h1 ~ h2 {
    color: yellow;
    background-color: aqua;
}
```

We can add background color the `<section>` element to create a box-like effect. The code below uses HSL and RGB, as an example.

```
section {
    background-color: hsl(360, 100%, 60%);
    color: rgb(255,241,0,0.5);
}
```

In practice, it's certainly fine to stick to one method (keywords, RGB, HSL, hexadecimal), or mix it up.

Data Types and Units of Measurements

Like many programming languages, CSS accepts a variety of data types. The common data types in CSS include:

- Integers: whole numbers (e.g., 1, 42, -5)
- Floating point, aka, decimal numbers (e.g., 1.5, 3.14, -0.75)
- Dimensional values: length, width, angle, and more
- Percentages: fractions represented as percentages

These data types are used primarily to add distance or length to elements. There are two broad categories of length and each include a variety of units:

- Relative lengths:
    - Font-relative lengths:
        - `em` : relative to the `font-size` of the parent element.
        - `rem` : relative to the `font-size` of the `<html>` element (root). Defaults to `16px` unless overridden.
    - Viewport-relative lengths:
        - `vw`
        - `vh`
        - `vmin`
        - `vmax`
- Absolute lengths:
    - `cm` : centimeters

- `mm` : millimeters
- `Q` : quarter-millimeters
- `in` : inches
- `pt` : points (1/72nd of 1 inch)
- `pc` : picas (1/6th of 1 inch)
- `px` : pixels (1/96th of 1 inch)

For the most part, we will rely on relative lengths to adjust fonts and other elements in our HTML pages. This is because relative lengths are more responsive to different device sizes and screen resolutions. However, it's important to know that absolute lengths exist if needed.

Consider the following HTML snippet. In the snippet below, I use internal CSS for demonstration purposes, but in practice, use an external stylesheet.

```
<html>
    <head>
        <title>sizes</title>
<style>
html {
    font-size: 1rem;
}

header {
    font-size: 1em;
}

h1 {
    font-size: 2em;
}

h2 {
    font-size: 2em;
}

p {
}

ul {
    font-size: 2em;
}

li {
    font-size: 1em;
}
</style>
    </head>
    <body>
        <header>
            <h1>header 1</h1>
            <h2>header 2</h2>
        </header>
        <p>This is a test.
        This is only a test.
        That is all.</p>
        <ul>
            <li>item 1</li>
            <li>item 2</li>
            <li>item 3</li>
            <li>item 4</li>
        </ul>
    </body>
</html>
```

Set `rem` in the `html` selector

```
html {
    font-size: 1rem;
    }
```

By default, browsers set `1rem = 16px`, unless overridden. In this example, we make the connection explicit.

Example: using `em` in `header`, `h1`, and `h2`

In the HTML example, note that `<header>` is a descendant element of the `<html>` element, and that `<h1>` and `<h2>` are siblings and both descendants of `<header>`. Now consider the following CSS:

```
header {
    font-size: 1em;
}

h1 {
    font-size: 1em;
}

h2 {
    font-size: 1em;
}
```

Since the `header` selector has a font-size of `1em`, it inherits its font size from the `html` selector since it's a descendant. Therefore, since `1em = 16px`, the `<header>` element is set to `16px`.

Since `<h1>` is inside `<header>`, and since it uses an `em` unit of measurement, it inherits its size from the `<header>` element. Since the `h1` selector is set to `2em`, then `2em * 16px = 32px`.

Since `<h2>` is also inside `<header>` (it is not a descendant but a sibling to `<h1>`), it also inherits from `<header>`. Since the `h2` is set to `2em`, then as with `h1`, we have `2em * 16px = 32px`. Specifically, since `header` is set to `1em`, its font size is unaffected with respect to `html`. Therefore, `h1` and `h2` are both calculated relative to `html`.

Example: Using `em` in `ul` and `li`

In the above HTML, the `<ul>` element is a descendant of `<html>`. Since `ul` is set to `2em`, then like above, `2em * 16px = 32px`. However, the `<li>` element is a child of `<ul>` and since its

font-size uses the `em` unit, then it inherits from `<ul>`. Since `ul` is calculated to be `32px`, then `li` is `1em * 32px = 32px`.

In practice, therefore, we can see how `em` compounds based on what it inherits from its parent element. This makes using `rem` more predictable. If we set the font-size of all elements to `1rem`, then the font-size for all elements would be `16px` because `1rem * 16px = 16px`. However, if we only set the `html` selector to `1rem`, then the browser uses its default sizes.

Be sure to understand inheritance.

Consider the following rules:

```
html {
    font-size: 1rem;
}

header {
    font-size: 2rem;
}

h1 {
    font-size: 1rem;
}

h2 {
    font-size: 1.5rem;
}
```

We might think that since `h1` and `h2` are child elements of `header`, this would result in very large font sizes for `<h1>` and `<h2>`. Remember though that `rem` is always relative to the font-size set in the `html` selector (typically `16px`). Even though the `header` is set to `2rem`, the `h1` and `h2` rules explicitly define their font sizes using `rem`. So they ignore the inherited value from `header`. That is, `h1` and `h2` are calculated with respect to `html`.

- `html`：`1rem * 16px = 16px`
- `h1`：`1rem * 16px = 16px`
- `h2`：`1.5rem * 16px = 24px`

However, if we only use the following two rules in our CSS:

```
html {
    font-size: 1rem;
}

header {
    font-size: 2rem;
}
```

Then both `h1` and `h2` would increase by `2rem` since they are direct children of `<header>`.

Page Sizing

Viewport relative lengths are useful when controlling the width of the sections on a page, like `<main>`, `<section>`, `<article>`, and like. Consider the following HTML snippet:

```
<html>
    <head>
        <title>sizes</title>
<style>

header, section {
    width: 50vw;
    margin: auto;
}

header {
    text-align: center;
}
</style>
    </head>
    <body>
        <header>
            <h1><cite>Their Eyes Were Watching God</cite></h1>
            <h2>Zora Neale Hurston</h2>
        </header>
        <section>
            <p><q>Ships at a distance have every man's wish on board. For some
they
            come in with the tide. For others they sail forever on the horizon,
            never out of sight, never landing until the Watcher turns his eyes
            away in resignation, his dreams mocked to death by Time. That is
            the life of men.</q></p>

            <p><q>Now, women forget all those things they don't want to remember,
and
            remember everything they don't want to forget. The dream is the
            truth. Then they act and do things accordingly.</q></p>
        </section>
    </body>
</html>
```

In the above code, I set the width of the `header` and `section` selectors to 50% of the viewport: `width: 50vw;`. Since `vw` (viewport width) is a relative unit, the elements inside these sections will automatically adjust when the browser window is resized.

I also set `margin: auto;`. The result is that the left and right margins will automatically adjust to ensure equal spacing on both sides and will be centered within the page.

I also used the `text-align: center;` for the `header` selector. This centers all text in the `<header>` element, which includes the `<h1>` and `<h2>` elements. Even though the `header` is only 50vw wide, the text inside is aligned to the center of that space.

Conclusion

It's important to understand the color systems and measurement units in CSS. We use these CSS technologies to create visually appealing and responsive designs. To recall:

- Color values like RGB, HSL, and hexadecimal allow precise control over appearance. Color keywords allow for easy color choices of common colors.
- Measurement units like `rem`, `em`, and viewport units help ensure layouts scale properly across different devices.
- The choice between relative (`em`, `rem`, `vw`) and absolute (`px`, `cm`, `in`) units depends on whether you need flexibility or fixed sizing.

When you understand these concepts, you will be able to build adaptable, maintainable, and user-friendly web designs. We'll explore more responsive techniques later in this work when we tackle Flexbox and Grid Layout methods.

The Box Model

The basic principle of working with CSS and HTML is knowing that all HTML elements are contained in a box. This includes everything from the root element `<html>` all the way down to text-level elements such as the paragraph tag `<p>` or the `<span>` tag. We can test this using the universal selector `*` in our CSS. For example, consider the following HTML, which quotes the first sentence from *Moby-Dick*:

```
<html>
    <head>
    </head>
    <body>
        <h1><cite><span class="all_cap">Moby-Dick;</span> or <span
class="all_cap">The Whale</span></cite></h1>
        <h2>By Herman Melville</h2>
        <p><q>Call me Ishmael.
            Some years ago&mdash;never mind how long precisely&mdash;having little
or no money in my purse,
            and nothing particular to interest me on shore,
            I thought I would sail about a little and see the watery part of the
world.</q></p>
        <p>Read the full text at: <a
href="https://www.gutenberg.org/ebooks/2701">Project Gutenberg</a></p>
    </body>
</html>
```

In the following snippet, I add the universal selector to place a box around all elements in the above web document.

```
<html>
    <head>
<style>
* {
    border: 1px solid black;
}
.all_cap {
    text-transform: uppercase;
}
</style>
    </head>
    <body>
        <h1><cite><span class="all_cap">Moby-Dick;</span> or <span
class="all_cap">The Whale</span></cite></h1>
        <h2>By Herman Melville</h2>
        <p><q>Call me Ishmael.
            Some years ago&mdash;never mind how long precisely&mdash;having little
or no money in my purse,
            and nothing particular to interest me on shore,
            I thought I would sail about a little and see the watery part of the
world.</q></p>
        <p>Read the full text at: <a
href="https://www.gutenberg.org/ebooks/2701">Project Gutenberg</a></p>
    </body>
</html>
```

Understanding this idea, that CSS treats HTML elements as boxes, is important for styling and designing layouts to our HTML. It's also important to understand that since boxes can be nested within other boxes, or sit adjacent to other boxes (e.g., descendant elements, sibling elements, etc), that styling elements has ripple effects. Adjusting a box's size or position affects surrounding elements, as boxes influence each other's layout. Therefore, to fully fully understand how to manipulate HTML elements, we need to understand that CSS boxes possess several properties.

Box Properties

The CSS box model consists of four main properties: the **content**, surrounded by **padding**, a **border**, and an outer **margin**. Figure 1 below visually represents these properties in the standard CSS box model.

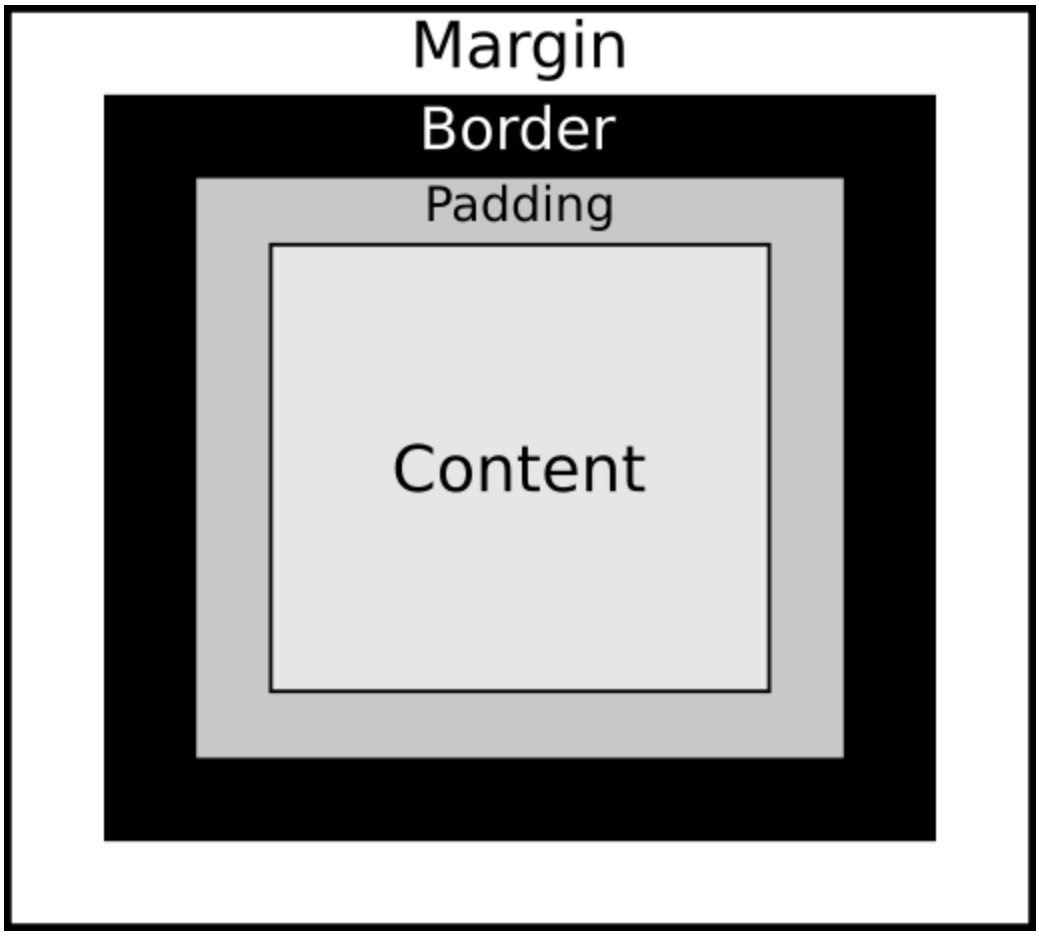


Fig. 1. The CSS Box Model

In CSS, we can style the margin, border, padding, and content. We have already discussed styling **content**. We do this when we, for example, change the font-size of an element, like using `em` and other units of measurement.

Conclusion

Mastering the box model takes time, as it serves as the foundation of CSS layout. Every adjustment has ripple effects for descendant and sibling elements. In the next section, we learn how to control a box's sizing and appearance by styling its margins, borders, and padding.

Margins, Borders, Padding, and Sizing

As you recall, the CSS box model contains four different properties.

- **margin**: the space between the element's border and its surrounding elements, marked by the margin edge
- **border**: the boundary between the margin and padding, which can made visual with width, color, and style properties
- **padding**: the space between the border and content
- **content**: the content

These properties are illustrated in the figure below.

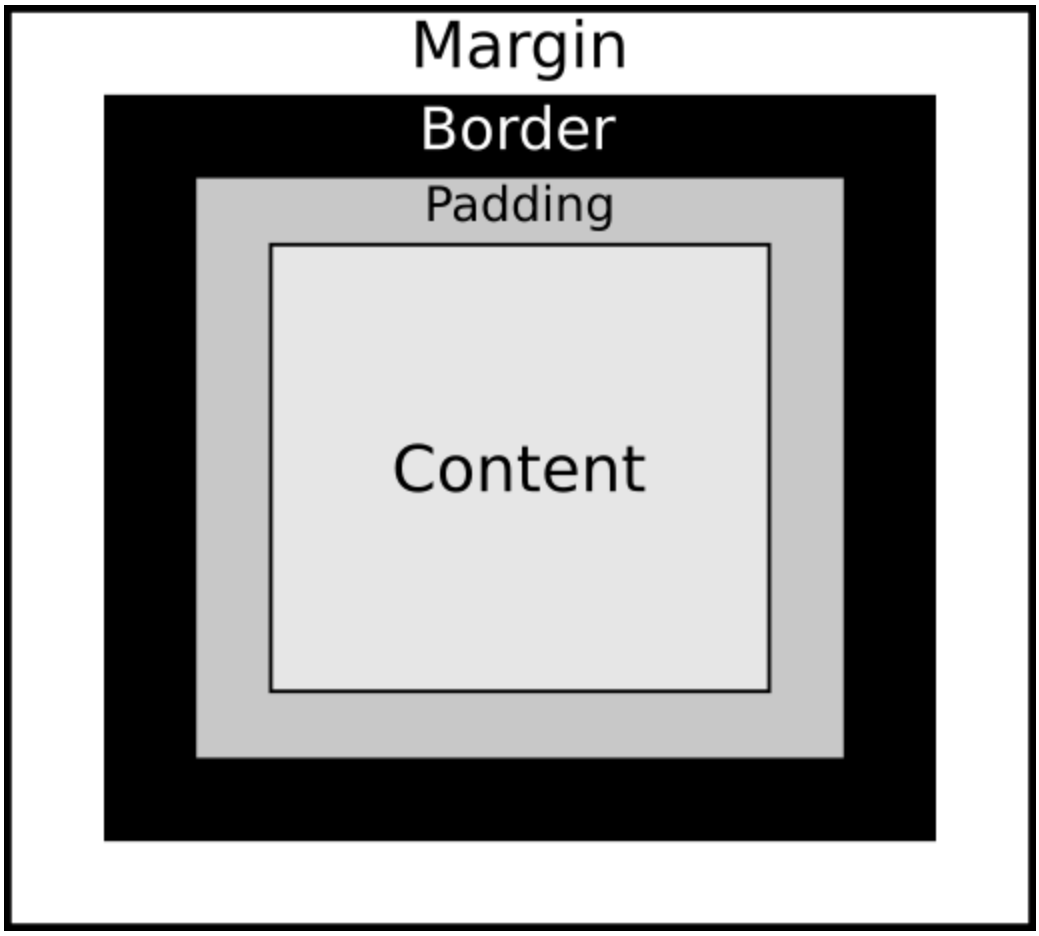


Fig. 1. The CSS Box Model

A key characteristic of the CSS box is that, like boxes in general, it has four edges, marked by the following keywords: **top**, **right**, **bottom**, and **left** (in that order). We can use CSS to target any of these sides, or all sides at once, and style boxes using the lessons covered in the prior section on Colors and Units of Measurement.

Margin

Let's start with some basic HTML that contains only a `<div>` element in the `<body>`:

```
<html>
    <head>
    </head>
    <body>
        <div>
            Building a basic box.
        </div>
    </body>
</html>
```

First, we'll add a border around the box, in order to highlight the `<div>` 's boundary between its padding and margin:

```
<html>
    <head>
<style>

div {
    border: 1px solid black;
}

</style>
    </head>
    <body>
        <div>
            Building a basic box.
        </div>
    </body>
</html>
```

Next, I add the `margin` property with a value of `2em` to demonstrate how it adjusts the spacing around the `<div>` element, pushing it away from adjacent elements. Since the default value of `em` is set by the browser to `16px`, using `2em` means the box will be offset from adjacent elements by 32 pixels on all four sides:

```
div {
    border: 1px solid black;
    margin: 2em;
}
```

Dimensions

The four edges of a box—top, right, bottom, left (in that order)—can be set using a single shorthand property. This shorthand works for the `margin` and `padding` properties. In the following example, I set the `margin-top` to `2em`, the `margin-right` to `3em`, the `margin-bottom` to `1em`, and the `margin-left` to `4em`.

```
<html>
    <head>
<style>

div {
    border: 1px solid black;
    /* top, right, bottom, left */
    margin: 2em 3em 1em 4em;
}

</style>
    </head>
    <body>
        <h1>Heading 1</h1>
        <div>Hello world</div>
        <h2>Heading 1</h2>
    </body>
</html>
```

Centering Boxes

To center a box horizontally, set a `width` and use `margin: auto;`. This distributes the remaining space equally on both sides and centers the element within its parent container. In the following snippet, I set the width of the `<div>` element to 90% of the window. Then by setting `margin: auto;`, the `<div>` element gets centered in the viewing area.

```
<html>
    <head>
<style>

div {
    border: 1px solid black;
    width: 90%;
    margin: auto;
}

</style>
    </head>
    <body>
        <h1>Heading 1</h1>
        <div>Hello world</div>
        <h2>Heading 1</h2>
    </body>
</html>
```

Note that we can also use `vw` to set the width. For example, instead of using `width: 90%;` we can use `width: 50vw;`.

Border

The above snippet introduces the `border` property. This property is shorthand for `border-width`, `border-style`, and `border-color`, each of which can be set separately. There is also a `border-radius` property that can be used to create rounded edges.

Borders may be styled with various colors (`rgba`, `hsl`, hexadecimal, or color keywords) and different styles, such as `solid`, `dotted`, `dashed`, `double`, and more. Some styles, like `groove`, `ridge`, `inset`, and `outset`, create three-dimensional effects. In the following snippet, I focus on the border's bottom edge by setting only the bottom's width, style, and color:

```
div {
    border-bottom-width: 10px;
    border-bottom-style: dashed;
    border-bottom-color: hsl(291,100%,50%);
    margin: 2em;
}
```

Padding

The `padding` property covers the space between the `border` and the `content`. Note that the effect is different when using relative lengths, like `em` or `rem`, and absolute lengths, like `px`. As you recall, the `em` is relative to the `font-size` of the parent element and `rem` is relative to the `font-size` of the `<html>` root element. Thus, the following three examples result in different box sizes of the `<div>` element.

Setting in Pixels

In the snippet below, the values for the `margin` and `padding` properties are set to 10 pixels regardless of the value of the `html` font size or the `<div>` 's parent element.

```
<html>
    <head>
<style>

html {
    font-size: 1rem;
}

div {
    border: 1px solid black;
    margin: 10px;
    padding: 10px;
}

</style>
    </head>
    <body>
        <div>
            Building a basic box.
        </div>
    </body>
</html>
```

Setting in `em` (relative to parent element)

In the snippet below, the values for the `margin` and `padding` properties are set to `2em`. Since the `<body>` element's `font-size` is `20px`, and `em` is relative to its parent, setting `margin: 2em;` results in `2 * 20px = 40px`.

```
<html>
    <head>
<style>

body {
    font-size: 20px;
}
div {
    border: 1px solid black;
    margin: 2em;
    padding: 2em;
}

</style>
    </head>
    <body>
        <div>
            Building a basic box.
        </div>
    </body>
</html>
```

You can test the relationship between the `<div>` and its parent element `<body>` by changing the font-size for the `body` selector.

Setting in `rem` (relative to root element)

In the following snippet, the font-size for the root `<html>` element is set to the default, which is `16px` (`1rem = 16px`). Since the `margin` and `padding` for the `div` selector is set to `2rem` each, then they will each be `32px` (`2rem * 16px = 32px`).

```
<html>
    <head>
<style>

html {
    font-size: 1rem;
}
div {
    border: 1px solid black;
    margin: 2rem;
    padding: 2rem;
}

</style>
    </head>
    <body>
        <div>
            Building a basic box.
        </div>
    </body>
</html>
```

Mixing `em` and `rem`

In the snippet below, the `p` selector uses `rem` units, and therefore its size is calculated with respect to the root element. However, the `b` selector (for the `<b>` element in the HTML section) uses `em` units. Since the `em` unit is calculated with respect to the parent element, and since the parent element in this case is the `<p>` element, then the size of the text enclosed in the `<b>` element is calculated with respect to the `<p>` element.

```
<html>
    <head>
<style>

html {
    font-size: 1rem;
}

p {
    border: 1px solid black;
    margin: 2rem;
    padding: 2rem;
}

b {
    border: 1px solid black;
    margin: 2em;
    padding: 2em;
}

</style>
    </head>
    <body>
        <p>This is <b>bold.</b></p>
    </body>
</html>
```

Content

We have already discussed the last property: **content**. For example, increasing `font-size` expands the content area, causing the box to grow accordingly. In the following code snippet, I have an `<h1>` element with some simple text:

```
<html>
    <body>
        <section>
            <h1>Heading 1</h1>
        </section>
    </body>
</html>
```

Now let's add a border around that. This reveals how the element is wrapped in a box:

```
<html>
    <head>
<style>

h1 {
    border: 1px solid black;
}

</style>
    </head>
    <body>
        <section>
            <h1>Heading 1</h1>
        </section>
    </body>
</html>
```

Increasing the `font-size` enlarges the content, which in turn expands the overall box dimensions.

```
<html>
    <head>
<style>

h1 {
    border: 1px solid black;
    font-size: 5em;
}

</style>
    </head>
    <body>
        <section>
            <h1>Heading 1</h1>
        </section>
    </body>
</html>
```

Box Sizing

When we set the dimensions of a box in CSS, the dimensions are calculated based on the size of the content and not on the dimensions (height, width) of the box's border or padding. The result is that “if you set an element's width to 100 pixels, then the element's content box will be 100 pixels wide, and the width of any border or padding will be added to the final rendered width, making the element wider than 100px” (The CSS `box-sizing` property).

The `border-box` property accounts for border and padding. In the following example, I nest a `<div>` within a `<div>` and add padding around the nested `<div>`. The width is set to `100%`, which intuitively suggests that the smaller box should fit within 100% of the larger box. But by adding padding (`10px`), the box expands beyond the larger box's border.

```
<html>
    <head>
<style>
.big, .small {
    border: 10px solid black;
    font-size: 3rem;
}

.big {
    width: 300px;
}

.small {
    padding: 10px;
    width: 100%;
}
</style>
    </head>
    <body>
        <div class="big">big box
            <div class="small">little box</div>
        </div>
    </body>
</html>
```

The result becomes more intuitive when adding `box-sizing: border-box;` to the small `<div>`. In the example below, the padding and border are adjusted to fit within the larger `<div>`. That is, the border width and padding, and not simply the content, are taken into account when sizing the box.

```
<html>
    <head>
<style>
.big, .small {
    border: 10px solid black;
    font-size: 3rem;
}

.big {
    width: 300px;
}

.small {
    box-sizing: border-box;
    padding: 10px;
    width: 100%;
}
</style>
    </head>
    <body>
        <div class="big">big box
            <div class="small">little box</div>
        </div>
    </body>
</html>
```

Conclusion

Understanding the CSS box model is essential for controlling layout and spacing of elements on a webpage. Remember that the **margin** creates space around an element, **borders** define its visible boundaries, **padding** controls the space between the border and content, and **content** itself determines the element's size. When we use these properties effectively, we can adjust positioning, control whitespace, and create visually appealing layouts.

Learning how the interactions among these properties work takes practice, especially when mixing units like `px`, `em`, and `rem`, or using various combinators. I encourage you to experiment with margin, padding, border, and content settings. As you do, you will gain an increasing understanding of how boxes work and interact in CSS.

Typography and Font Styles

The *typography* of a document is generally its most visible aspect. This is true regardless if the document is a web page, a book, a billboard, or a flyer. We use typographic principles to shape how text is presented, read, and interpreted.

In web development, typography is important because it influences the user experience. It affects readability, aesthetics, and accessibility, and it plays a central role in how content is presented across different screen sizes and devices.

In all, good typography can:

- improve readability and legibility
- establish identity and tone
- enhance user engagement
- maximize accessibility for users, and this includes those with visual impairments
- contribute to an intuitive and pleasant user experience

Key Concepts in Typography

Typography consists of several elements that determine how text appears. These elements include:

- `font-family` : this is the specific typeface or group of typefaces. Major typefaces include: `serif` , `sans-serif` , `monospace`
- `font-style` : this includes variations like normal, italic, or oblique
- `font-weight` : this addresses the thickness of characters and ranges from light to bold
- `font-size` : this consists of the height of characters. It can be set in fixed or relative units, and it alters the content-size of its box
- `letter-spacing` , or kerning: this addresses the spacing between individual letters
- `line-height` : this addresses the spacing between lines of text
- `text-align` : this addresses the horizontal alignment of text and is used to center or justify text
- `text-indent` : this addresses the indentation of text

Typography vs. Fonts

Typography refers to the overall design and arrangement of text. It includes the choice of fonts, how fonts are spaced and aligned, and other readability considerations.

In CSS, and thus in practice, we largely work with properties related to **fonts**. Fonts are specific styles or variations of typefaces (aka, font families). In short, typography is the art of arranging text, while fonts are the specific tools used in that arrangement.

Basic Typography Principles

The goal of typography is to improve readability. This is accomplished by creating web pages and sites that are visually appealing. The following key principles help achieve that goal:

- **Readability and legibility**: Use appropriate font choices, sizes, and spacing to make sure that text is easy to read.
- **Hierarchy**: Implement variations in font size, weight, and color because it will help establish structure and guide the reader's attention.
- **Consistency**: Maintain uniformity in the typographic elements, such as font families, alignment, and spacing, you use to create a cohesive design.
- **Alignment and balance**: Arrange text in a consistent and structured way. This includes properties like left alignment, centering, and justifying. This will help maintain visual balance and improve reading comprehension.
- **Contrast**: Differentiate headings, body text, and other elements through the use of font weight, font size, and font color. This will help make content more engaging.
- **White space (or negative space)**: Use adequate spacing around text. This will help avoid clutter and improve readability.

Font Families

Of all the typographic elements above, the one with the most impact is the `font-family`. The major font families include `serif`, `sans-serif`, and `monospace`, but there are more. We select font families based on several criteria:

- **Serif fonts**: these are typefaces that have decorative strokes, or *serifs*, at the ends of letters. Common examples include Times New Roman and Georgia. These kinds of typefaces are commonly used in print and they convey a classic, professional look.
- **Sans-serif fonts**: these fonts lack (*sans*) serifs or decorations. These fonts convey cleaner and more modern designs. Common examples include Arial, Helvetica, and Verdana. They

benefit digital content because they augment clarity and readability on screens in particular.
- **Monospace fonts**: the characters in these fonts take up the same amount of horizontal space. Examples include Courier New and Consolas. These font families are typically used in coding environments or to display code in web documents because they make it easier to align characters and read structured text.

Typography in Web Development

Web developers and designers use CSS to control typography across different devices and screen sizes. The goal is to always make sure that text is legible and visually appealing. While this section covers the basics, typography is a big subject. Entire books are dedicated to typography, and companies spend lots of resources on developing and using good fonts because fonts play an important role in brand identity. However, as you begin to use and then understand typography principles, you will be able to use these principles to augment content and enhance user experience.

Typography in Practice

Font Family

We normally declare the `font-family` in the `html` or `body` selector. When we declare a `font-family`, we provide a list of families. Based on our operating systems, our browsers select the font to display by first available. Therefore, we place the main, desired font at the beginning of the list but then provide fallback fonts in case the main font is unavailable in the reader's browser.

In the following example, `Arial` is the font we ideally want to use in our web document, but if this isn't available, then the browser falls back to the generic `sans-serif` font family. For consistency, be sure to include fonts from the same family in this list (e.g., all serif, all sans-serif, or all monospace but not a mix).

```
body {
    font-family: "Arial", sans-serif;
}
```

Note on Examples Below

In the examples below, I use classes to demonstrate how to implement font and other properties. In the first example, I use the class selector `.italic-text`. This is implemented in HTML with the `class` attribute. For example, if I want to italicize several words in a paragraph in my HTML, I could use a `<span>` element with a `class` attribute of `italic-text`:

```
<p>This <span class="italic-text">text is italicized</span>.</p>
```

This would render as:

This *text is italicized.*

Font Styles

We use the `font-style` to designate whether a font should be `normal`, `italic`, or `oblique`.

```
.italic-text {
    font-style: italic;
}

.oblique-text {
    font-style: oblique;
}
```

We can also control the slope of the font by setting a value for degrees (`deg`):

```
.oblique-text {
    font-style: oblique 10deg;
}
```

Note that in HTML, we can use the `<i>` or `<em>` elements to add some kind of emphasis to our text that generally renders it as italicized text. It's generally worthwhile to use these HTML elements when the reason for the emphasis has a semantic purpose.

Font Weight

We use the `font-weight` property to thickness of the font, or its boldness. Since font thickness is defined numerically, we can specify a range of weights, from light to heavy.

```
.light-text {
    font-weight: 200; /* Thin */
}

.bold-text {
    font-weight: bold; /* Equivalent to 700 */
}

.extra-bold-text {
    font-weight: 900; /* Heavy weight */
}
```

In HTML, we can use the `<b>` or `<strong>` elements to render text as bold. Again, unless there is some semantic reason to do so, use the CSS `font-weight` property instead.

Font Size and Scaling

It's a good idea to clarify the default `font-size` in the `html` selector, and then modify font-sizes throughout the document with respect to that.

```
html {
    font-size: 1rem;
}
```

However, we can also implement different font-sizes throughout the document:

```
.normal-text {
    font-size: 16px; /* Fixed size */
}

.relative-text {
    font-size: 1.2em; /* Relative to parent element */
}

.responsive-text {
    font-size: 3vw; /* Viewport width-based */
}
```

Letter and Word Spacing

We have two properties to control the spacing between either letters or words. This kind of property, generally referred to as kerning, can augment readability. However, be careful implementing this property. Most typefaces have been designed with default kerning parameters, and the basic font families have been well tested for readability.

```
.kerning-example {
    letter-spacing: 2px; /* Adjusts letter spacing */
}

.word-spacing-example {
    word-spacing: 5px; /* Adjusts space between words */
}
```

Line Properties and Text Layout

The `line-height` property adjusts the spacing between lines. Like kerning, be conservative when applying this property since it may impact legibility. In the following example, we use a class (`.line-height-example`), but this for the kind of property, we might want to apply it consistently by applying to one specific element, such as the `<p>` element.

```
.line-height-example {
    line-height: 1.5; /* Adjusts leading */
}
```

We can `center` or `justify` text with the `text-align` property:

```
.centered-text {
    text-align: center;
}

.justified-text {
    text-align: justify;
}
```

We can use the `text-indent` property to indent only the first line of an element. This is especially useful for paragraph `<p>` elements, but a good use case is to use it as a class in a `<span>` element and indent the first line of a paragraph immediately after a heading (e.g., `<h2>`) element:

```
.text-indent-example {
    text-indent: 20px; /* Indents first line */
}
```

Text Decoration and Transformation

The `text-decoration` property adds decorative lines to our text, and the `text-transform` property controls a text's capitalization.

Since `text-decoration` adds a line, the line may be styled as `solid`, `double`, `dotted`, `dashed`, or `wavy`. The default is to underline text, but we can also use `text-decoration` to add an overline or line-through. We may also color the line.

```
.underline-text {
    text-decoration: underline;
}

.strike-through-text {
    text-decoration: line-through;
}

.multi_decoration {
    text-decoration: overline double green 5px;
}
```

The `text-transform` takes several values for capitalization, as demonstrated below. A good use case is to apply this to various heading elements (e.g., `<h2>`), such as making all such elements uppercase:

```
.all-uppercase {
    text-transform: uppercase;
}

.all-lowercase {
    text-transform: lowercase;
}

.capitalize-text {
    text-transform: capitalize;
}
```

Outsourcing Fonts

As stated before, fonts are generally provided by the operating system. Since there are different operating systems (e.g., Windows, macOS, iOS, Linux, Android, etc), not all fonts are available for all OSes. OSes from Microsoft, Apple, and Google might share the most common fonts because commercial companies pay or charge licensing fees for many standard fonts since fonts may be copyrighted. In any case, this is why when we declare font families, we always declare `serif` as the fallback font-family if we are using a copyrighted serif font like Times New Roman; or `sans-serif` as the fallback font-family if we are using a copyrighted sans-serif font like Arial; and so forth.

However, we can load fonts from external sources. One major example is Google Fonts. For example, if we want to use the Roboto Font, its Google Font page provides the code to insert into our HTML pages and CSS stylesheets.

Locally Hosted Fonts

Outsourced fonts present several issues. First, we may not know or agree with the privacy policies when using outsourced fonts. Also, outsourcing fonts means relying on third-party services. While these services may be reliable, they could go down. Lastly, depending on the source used for outsourced fonts, using outsourced fonts may requires extra bandwidth:

> The Google Fonts Privacy Policy makes an argument that using their services instead of self-hosting their fonts saves bandwidth.

Fortunately, we can also store and host fonts locally from within our project directory. This offers greater control and performance and allows our desired fonts to be used regardless of a user's OS or browser. To do this, we need to:

1. Download the font files:
    1. we obtain font files in formats like `.woff`, `woff2`, `ttf`, or `.otf` from a **trusted** source.
    2. we save those fonts to a designated `fonts/` directory in our project.
2. Define the font in our CSS:
    1. we use the `@font-face` rule to specify the font name and file path.

Finally, we might add the following to our CSS stylesheet to use a font named `CustomFont`:

```
@font-face {
    font-family: 'CustomFont';
    src: url('fonts/CustomFont.woff2') format('woff2'),
         url('fonts/CustomFont.woff') format('woff');
    font-weight: normal;
    font-style: normal;
}

body {
    font-family: 'CustomFont', sans-serif;
}
```

Note: Although the above example illustrates how to use downloaded fonts, since the `src` property takes a `url` value, you can use the `@font-face` rule to source remote font files. See the documentation for the `@font-face` rule for more details.

If using locally hosted fonts, be sure you use fonts from **trusted** sources and fonts that have open licenses. Just like images, text, etc, fonts can be copyrighted. I have not vetted all these sources, but some options you may explore for locating fonts to download and use on your sites include:

- Font Squirrel
- The League of Moveable Type
- Adobe Fonts
- Velvetyne Type Foundry
- Open Foundry
- Font Library
- 1001 Fonts

Conclusion

Typography plays a crucial role in shaping how content is presented, read, and understood. Try to select the right font family for your site visitors. Keep those visitors in mind, always!

And be sure to use fine-tuning properties, like spacing, weight, and alignment, to enhance aesthetics and readability.

Finally, focus on the principles: readability, legibility, hierarchy, consistency, alignment and balance, contrast, and white space.

Styling Images, Lists, and Backgrounds

Introduction

We have discussed how to add images to our web pages. Now it's time to learn to style them. We will also learn how to style lists, which will include adding images to our lists, and styling the background canvas of a web page. For the full documentation, see CSS Images Module Level 3.

Prerequisites

When we add images to a web page, we want to make sure that our images have been properly sized in a separate photo editor. For example, if I add a photo to a web page that has the resolution of something like 1920x1080 pixels, then the photo may be several megabytes in size. A photo this large uses more bandwidth, which may slow down how quickly the web page displays in the browser. Therefore, before we tinker with sizes in HTML or CSS, we need to reduce the resolution of our images in a separate photo editor.

Second, we may want to change the default format of our images. Most of our cameras will default to saving photos as JPEGs. Or we might generate images in PNG format. These are both popular image formats that are used on the web, along with GIFs. However, another popular format is WEBP, an image format optimized for the web and that supports lossy and lossless compression, transparency, and animation. Photo and other types of image editors can export images to WEBP. For example, in the GIMP photo editor, I can export photos as WEBP and choose lossless or lossy compression when exporting.

Another excellent image format is SVG (Scalable Vector Graphics). SVG formats are used for vector based images and are the default file format for programs like Inkscape. These images are ideal for images that are drawn, like logos, icons, or similar forms of visual art. Also, since SVG files are text files, we can use scripting languages, like Python, to modify them. But one of the most important characteristics of SVG files is that they are scalable. That is, no matter how much we zoom in or out on a SVG file, the image will retain its resolution, unlike raster or bitmap based images, which will become pixelated.

Images

In HTML and CSS, we can set the values of an image with simple `width` and `height` settings. For example, in HTML, we can use the `width` and `height` attributes:

```
<img src="media/walking_bridge.jpg"
    alt="A walking bridge over a stream"
    width="673"
    height="378">
```

Alternatively, we can leave out the HTML attributes in the above code and instead add these properties to our CSS. In the CSS below, I use the `img` selector with the specified `width` and `height` properties:

```
img {
    width: 673px;
    height: 378px;
}
```

It's important to understand that resizing images directly in HTML or CSS does not reduce the image's **file size**. It only reduces the **display size**. Therefore, be sure to properly resize the image using a photo editor to optimize your web page's performance.

`object-fit`

Recall that the CSS box model informs everything about how content is styled on a page. This means that even images are enclosed within boxes. The following HTML code adds a simple drawing to our web page, and the CSS code sets a border, background color, and dimensions.

HTML:

```
<img src='media/box_circle.png' alt='A drawing of a box in a circle'>
```

CSS:

```
img {
    border: 1px solid black;
    background-color: silver;
    width: 300px;
    height: 400px;
}
```

Using the `object-fit` property, we can scale the image within the box. Values include `fill`, `contain`, `cover`, `none`, and `scale-down`. First, in the HTML code below, I use classes to style the same image for each of those values.

```
<h3>Fill</h3>
<img class="fill" src='media/box_circle.png' alt='A drawing of a box in a circle'>

<h3>Contain</h3>
<img class="contain" src='media/box_circle.png' alt='A drawing of a box in a
circle'>

<h3>Cover</h3>
<img class="cover" src='media/box_circle.png' alt='A drawing of a box in a
circle'>

<h3>None</h3>
<img class="none" src='media/box_circle.png' alt='A drawing of a box in a circle'>

<h3>Scale Down</h3>
<img class="scale-down" src='media/box_circle.png' alt='A drawing of a box in a
circle'>
```

In the CSS code below, I apply the `object-fit` property to the appropriate classes in the above HTML.

The original image (`box_image.png`) has the dimensions of `513x503`, but we set the `width: 300px` and `height: 400px` in the `img` selector. Therefore, the `fill` value will distort the image so that it fills the box:

```
.fill {
    object-fit: fill;
}
```

The `contain` value scales the content so that it maintains its original aspect ratio:

```
.contain {
    object-fit: contain;
}
```

The `cover` value scales the content so the content completely covers the box and preserves the original aspect ratio:

```
.cover {
    object-fit: cover;
}
```

The `none` value displays the content at its original size (513x503):

```
.none {
    object-fit: none;
}
```

The `scale-down` value displays the content at its smallest possible size between `none` and `contain`:

```
.scale-down {
    object-fit: scale-down;
}
```

Position

Images may also be positioned within their surrounding boxes using the `object-position` property. By default, an image's position is centered in its box. We can adjust the x-axis and y-axis coordinates to move the image within its box. In the HTML below, I use classes to specify the position of the images: default, top, right, bottom, and left.

```
<h3>Image Default</h3>
<img class="position" src='media/box_circle.png' alt='A drawing of a box in a
circle'>

<h3>Image Top</h3>
<img class="position_top" src='media/box_circle.png' alt='A drawing of a box in a
circle'>

<h3>Image Right</h3>
<img class="position_right" src='media/box_circle.png' alt='A drawing of a box in
a circle'>

<h3>Image Bottom</h3>
<img class="position_bottom" src='media/box_circle.png' alt='A drawing of a box in
a circle'>

<h3>Image Left</h3>
<img class="position_left" src='media/box_circle.png' alt='A drawing of a box in a
circle'>
```

In the CSS below, I apply the classes by modifying the x-axis and y-axis for each:

```
.position {
    object-position: 50% 50%;
}
.position_top {
    object-position: 50% -50px;
}
.position_right {
    object-position: 50px 50%;
}
.position_bottom {
    object-position: 50% 50px;
}
.position_left {
    object-position: -50px 50%;
}
```

Transform

We can also transform images and "rotate, scale, skew, or translate" (MDN `transform`) them (and other elements, too). Consider the basic HTML code:

```
<img class="flip" src='media/box_circle.png' alt='A drawing of a box in a circle'>
```

Using the CSS `transform` property, we can rotate it by half a turn:

```
.flip {
    transform: rotate(0.5turn);
}
```

See the link above for examples of the other methods.

Floats

The `float` property lets us position images adjacent to text (or other elements). This technique is useful when creating layouts where text wraps around images.

Consider the following HTML examples. In each example, we add an `<img>` element adjacent to a `<p>` element. In the first example, we use the default behavior, which is to not `float` the image:

```
<h3>No Float (Default)</h3>
<img class="small" src='media/box_circle.png' alt='A drawing of a box in a
circle'>
<p>Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum. Excepteur sint occaecat
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est
laborum.</p>
```

In the next example, we `float` the image to the left of the text:

```
<h3>Float Left</h3>
<img class="left_float" src='media/box_circle.png' alt='A drawing of a box in a
circle'>
<p>Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum. Excepteur sint occaecat
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est
laborum.</p>
```

To create that effect, I create a class for this element and add `float: left` . I add a few pixels on the right margin of the `img` to add a gap between the image and the text.

```
.left_float {
    width: 75px;
    height: 75px;
    float: left;
    margin-right: 5px;
}
```

In the last example, we `float` the image to the right of the text:

```
<h3>Float Right</h3>
<img class="right_float" src='media/box_circle.png' alt='A drawing of a box in a
circle'>
<p> Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore
eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum. Excepteur sint occaecat
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est
laborum.</p>
```

Similar to the left float above, below the CSS simply floats the right using `float: right`. Also note that I change the margin side to left with `margin-left`:

```
.right_float {
    width: 75px;
    height: 75px;
    float: right;
    margin-left: 5px;
}
```

Gradients

We can use `gradient` colors to style images or backgrounds. We can apply gradients using angles (e.g., `45deg`) or keywords (e.g., `to top`). See the link above for other examples.

```
<h3>Linear Gradient</h3>
<img class="linear" src='media/box_circle.png' alt='A drawing of a box in a
circle'>

<h3>Linear Gradient Using Angle</h3>
<img class="angle" src='media/box_circle.png' alt='A drawing of a box in a
circle'>

<h3>Linear Gradient Using Keywords</h3>
<img class="top" src='media/box_circle.png' alt='A drawing of a box in a circle'>
```

In CSS below, I use a default `linear-gradient` in the first CSS class. In the second class, I use degrees to specify the exact gradient angle. In the third class, I use the `to top` keyword.

```
.linear {
    padding: 50px;
    background: linear-gradient(white, black);
}

.angle {
    padding: 50px;
    background: linear-gradient(45deg, white, black);
}

.top {
    padding: 50px;
    background: linear-gradient(to top, silver, green);
}
```

The main gradient keywords include:

- `to top` equates to `0deg`
- `to right` equates to `90deg`
- `to bottom` equates to `180deg`
- `to left` equates to `270deg`

Opacity

We can change the transparency of an image using the `opacity` property. A value of `0.0` indicates full transparency and a value of `1.0` indicates a fully opaque image.

HTML:

```
<h3>High Opacity</h3>
<img class="opacity_low" src='media/box_circle.png' alt='A drawing of a box in a
circle'>

<h3>Medium Opacity</h3>
<img class="opacity_med" src='media/box_circle.png' alt='A drawing of a box in a
circle'>
```

CSS:

```
.opacity_low {
    opacity: 0.2;
}

.opacity_med {
    opacity: 0.5;
}
```

Lists

We can create **ordered** or **unordered** lists using the `<ol>` and `<ul>` HTML elements. CSS offers flexibility in customizing lists. We can use images, emojis, or custom markers for list items.

Standard Lists

Let's begin with a standard unordered list:

```
<h3>Standard Unordered List</h3>
<ul>
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ul>
```

List Marker Positions

We can use the `list-style-position` to alter the position of the bullets. This also works for ordered lists using the `<ol>` element:

```
.inside {
    list-style-position: inside;
}
```

In the following HTML, I apply the position using a class:

```
<h3>Unordered List: Inside Position</h3>
<ul class="inside">
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ul>
```

Reverse Lists

In ordered lists, we can reverse the order of the counter using the `reverse` attribute (no CSS needed!):

```
<h3>Reversed Ordered List</h3>
<ol reversed>
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ol>
```

List Values

We can also begin counting at arbitrary numbers (whole numbers only, though):

```
<h3>New List Counter</h3>
<ol start="10">
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ol>
```

We can combine both attributes. In the following example, we start the count at `10` and then countdown using the `reverse` attribute:

```
<h3>Reversed New List Counter</h3>
<ol start="10" reversed>
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ol>
```

Roman Numerals

We can change to Roman numerals:

```
<h3>Upper Roman Numeral</h3>
<ul class="roman_upper">
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ul>

<h3>Lower Roman Numeral</h3>
<ul class="roman_lower">
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ul>
```

We define the use of Roman numerals in CSS. To use uppercase Roman numerals, we use `list-style-type: upper-roman`. To use lowercase Roman numerals, we use `list-style-type: lower-roman`.

```
.roman_upper {
    list-style-type: upper-roman;
}

.roman_lower {
    list-style-type: lower-roman;
}
```

English Characters

We can also use English characters by using either `list-style-type: lower-alpha` or `list-style-type: upper-alpha`:

HTML:

```
<h3>Lower Alpha</h3>
<ul class="alpha_lower">
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ul>

<h3>Upper Alpha</h3>
<ul class="alpha_upper">
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ul>
```

CSS:

```
.alpha_lower {
    list-style-type: lower-alpha;
}

.alpha_upper {
    list-style-type: upper-alpha;
}
```

Removing List Markers

We can remove all marks using `list-style-type: none`:

HTML:

```
<h3>No Marks</h3>
<ol class="no_marks">
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ol>
```

CSS:

```
.no_marks {
    list-style-type: none;
}
```

Changing List Markers

Or may also specify using a `circle`, a `disc`, Unicode, or other characters.

HTML:

```
<h3>Circle</h3>
<ol class="circle">
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ol>
```

CSS:

```
.circle {
    list-style-type: circle;
}
```

Emojis as List Markers

If we want to use emojis as our markers, then we need to use the corresponding Unicode for the emoji. See the Full Emoji List for ideas. In the following example, I use the Vulcan Salute as a marker for my list:

```
<h3>Vulcan Salute</h3>
<ol class="vulcan">
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ol>
```

CSS:

```
.vulcan {
    list-style-type: "\1F596";
}
```

Custom Images as List Markers

We can also use our own images. In the examples below, I used ChatGPT to generate icons for the fruit images. These icons were generated as `webp` files, and I refer to them using using the `url` value in the CSS.

In the first list below, I use one icon for all marks:

HTML:

```
<h3>Unordered List Using Images</h3>
<ul class="fruit_basket">
    <li>Apple</li>
    <li>Banana</li>
    <li>Watermelon</li>
</ul>
```

CSS:

```
.fruit_basket {
    list-style-image: url('media/fruit.webp');
}
```

In the following list, I use a separate fruit icon for each item in the list:

```
<h3>Unordered List Using Separate Images</h3>
<ul>
    <li class="apple">Apple</li>
    <li class="banana">Banana</li>
    <li class="watermelon">Watermelon</li>
</ul>
```

CSS:

```
.apple {
    list-style-image: url('media/apple.webp');
}

.banana {
    list-style-image: url('media/banana.webp');
}

.watermelon {
    list-style-image: url('media/watermelon.webp');
}
```

Nested Lists

Additionally, we can nest lists to create outlines or apply different `list-style-type` values for more variety.

Nested Ordered Lists

In the following example, I add an ordered list within an ordered list:

```
<h4>Ordered Nested</h4>
<ol>
    <li>Apples
        <ol>
            <li>HoneyCrisp</li>
            <li>Fuji</li>
        </ol>
    </li>
    <li>Bananas
        <ol>
            <li>Cavendish</li>
            <li>Plantain</li>
        </ol>
    </li>
    <li>Watermelon
        <ol>
            <li>Crimson Sweet</li>
            <li>Yellow Baby</li>
        </ol>
    </li>
</ol>
```

Nested Unordered Lists

Likewise, we can nest an unordered list within an unordered list:

```
<h4>Unordered Nested</h4>
<ul>
    <li>Apples
        <ul>
            <li>Honeycrisp</li>
            <li>Fuji</li>
        </ul>
    </li>
    <li>Bananas
        <ul>
            <li>Cavendish</li>
            <li>Plantain</li>
        </ul>
    </li>
    <li>Watermelon
        <ul>
            <li>Crimson Sweet</li>
            <li>Yellow Baby</li>
        </ul>
    </li>
</ul>
```

Nested Mixed Lists

We can also mix ordered and unordered lists when nesting. In the following two lists, I nest an unordered list within an ordered list. Then an ordered list within an unordered list.

Ordered to Unordered

```
<h4>Ordered/Unordered Nested</h4>
<ol>
    <li>Apples
        <ul>
            <li>Honeycrisp</li>
            <li>Fuji</li>
        </ul>
    </li>
    <li>Bananas
        <ul>
            <li>Cavendish</li>
            <li>Plantain</li>
        </ul>
    </li>
    <li>Watermelon
        <ul>
            <li>Crimson Sweet</li>
            <li>Yellow Baby</li>
        </ul>
    </li>
</ol>
```

Unordered to Ordered

In this example, I nest ordered lists in an unordered list:

```
<h4>Unordered/Ordered Nested</h4>
<ul>
    <li>Apples
        <ol>
            <li>Honeycrisp</li>
            <li>Fuji</li>
        </ol>
    </li>
    <li>Bananas
        <ol>
            <li>Cavendish</li>
            <li>Plantain</li>
        </ol>
    </li>
    <li>Watermelon
        <ol>
            <li>Crimson Sweet</li>
            <li>Yellow Baby</li>
        </ol>
    </li>
</ul>
```

Nested, Mixed Style Types

When nesting, we can apply a different `list-style-type` for the nested lists:

```
<h4>Roman Upper/Lower Nested</h4>
<ol class="roman_upper">
    <li>Apples
        <ul class="roman_lower">
            <li>Honeycrisp</li>
            <li>Fuji</li>
        </ul>
    </li>
    <li>Bananas
        <ul class="roman_lower">
            <li>Cavendish</li>
            <li>Plantain</li>
        </ul>
    </li>
    <li>Watermelon
        <ul>
            <li>Crimson Sweet</li>
            <li>Yellow Baby</li>
        </ul>
    </li>
</ol>
```

Styling the Background

We can set the background color of an HTML page by using the `background` property. For example, if we want a black background with white text, then we can set that using the `html` selector:

```
html {
    background: black;
    color: white;
}
```

But we have tools to offer more control using gradients. For example, we can create cool gradient effects with `linear-gradient`, `radial-gradient`, `conic-gradient`, and other properties. In the following example, I use the `linear-gradient` property to create a white and gray gradient across the page:

```
html {
    background: linear-gradient(white, gray);
}
```

And the `radial-gradient` property to create a yellow and green radial gradient across the page:

```
html {
    background: radial-gradient(yellow, green);
}
```

We can also use gradient keywords. In the example below, I create a radial gradient focused on the left bottom of the page:

```
html {
    background: radial-gradient(farthest-side at left bottom, yellow 50px, green);
}
```

When using images for the background, we use the `url` keyword with the location to the image. By default, background images repeat across the x-axis and y-axis. I use an image of a small grid or square to create a kind of notebook effect for the page:

```
html {
    background: url('media/small_grid.gif');
}
```

We need to use the `background-repeat: no-repeat` property and value to disble image repeating:

```
html {
    background: url('media/squiggly.png');
    background-repeat: no-repeat;
}
```

We can control the axis where the image repeats and the position on the page with the `repeat-x` value and `background-position: center;` declaration:

```
html {
    background: url('media/squiggly.png');
    background-repeat: repeat-x;
    background-position: center;
}
```

And the same for the y-axis:

```
html {
    background: url('media/squiggly.png');
    background-repeat: repeat-y;
    background-position: center;
}
```

We can also fix the image to a specific location so that the image remains fixed when the user scrolls the page:

```
html {
    background: url('media/squiggly.png');
    background-repeat: repeat-x;
    background-attachment: fixed;
}
```

Conclusion

In this section, we covered various CSS techniques for styling images, lists, and backgrounds. We manipulated images with properties like `object-fit` and `transform`, we customized lists with different styles and markers, and we also explored backgrounds with gradients, fixed positions, and background images. CSS offers a lot of flexibility.

Please experiment with these concepts to create visually engaging and user-friendly (or wild looking) web pages. Have fun, practice, and build your knowledge.

Appendix: Favicons

Your browser will display a favicon for a site in the site's tab, bookmark, etc. You can create favicons using most of the major file formats: PNG, JPEG, SVG. You can also use the ICO file format. The important thing to know is that favicons may be created in mutiple sizes and but should maintain a perfectly square ratio (1:1 aspect ratio). The most common sizes are 16x16 pixels, 32x32 pixels, 48x48 pixels, 96x96 pixels, etc. These sizes are used on different devices or under different views (tab, bookmark, etc). Google suggests using a favicon that's larger than 48x48px. However, it's common to make available multiple sizes and the browser can choose which one to use.

Favicons can be created in a photo editor, like GIMP, or better yet, in an SVG editor, like Inkscape. To use a favicon on a site, we use the `<link>` tag in the `<head>` of a web document. The following is an example of linking to four different files in a separate `icons/` directory with different favicon sizes:

```
<link rel="icon" type="icons/png" sizes="16x16" href="icons/icon_16.png">
<link rel="icon" type="icons/png" sizes="32x32" href="icons/icon_32.png">
<link rel="icon" type="icons/png" sizes="72x72" href="icons/icon_72.png">
<link rel="icon" type="icons/png" sizes="114x114" href="icons/icon_114.png">
<link rel="apple-touch-icon" sizes="144x144" href="icons/icon_144.png">
```

The files in the above code represent the different image sizes:

- `icon_16.png` : 16x16 pixels
- `icon_32.png` : 32x32 pixels
- `icon_72.png` : 72x72 pixels
- `icon_114.png` : 114x114 pixels

The `rel` attribute with the value of `apple-touch-icon` makes the favicon more accessible for iOS devices (because Apple).

Responsive Layouts with Flexbox

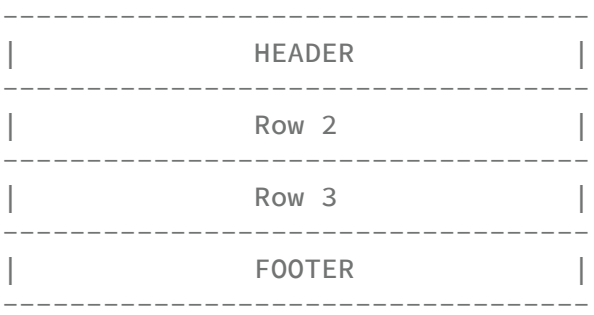
Introduction

Responsive design is of fundamental importance to the web. Since smartphones were introduced in 2007, smartphone browsers have increasingly become the dominant way to interact with the web. As a result of this trend in user behavior, search engines prioritize a smartphone-first web. Google, for example, stopped indexing sites that are not mobile accessible as of July 2024. This means that if your website isn't responsive, i.e., it doesn't collapse into a single column in mobile view, then it will eventually not show up in Google searches.

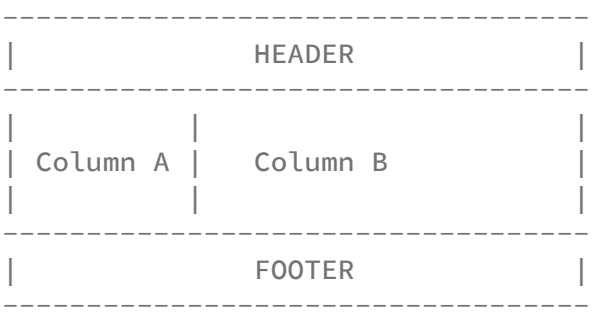
Common Layouts

So far in this work, we have primarily covered different ways to style the content of our web pages. However, one of the main purposes of CSS is to style the layout of a page. By far, the most common layouts include variations of the following:

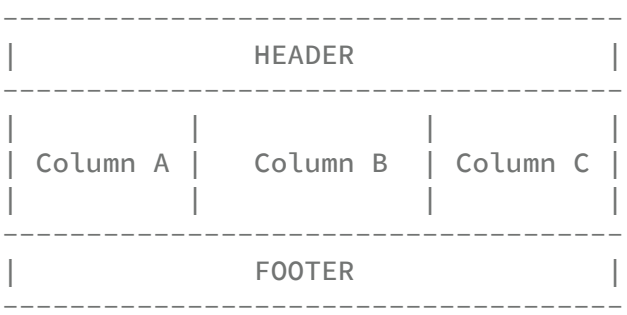
1-column layout



2-column layout

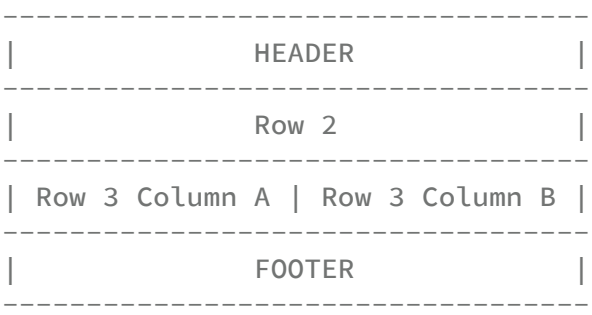


3-column layout



Nested Layout

Using more advanced techniques, we can nest layouts within layouts. For example, we can divide the body of a web page into additional grids, row- or column-wise. In the following example, the 1-column layout includes an extra column in the third row:



The MDN Web Docs provides a nice overview of the common layouts in web design.

Responsive Layouts

Layouts with more than one column only work on devices such as tablets, laptops, and desktop machines. When viewing on mobile, all layouts should be a 1-column layout.

Fortunately, CSS provides the tools to create **responsive** designs; i.e., tools needed to collapse multi-column layouts into a single layout. The two newest tools include **Flexbox** and **Grid**. Both of these tools make it much easier to create complex, multi-row and multi-column, layouts that also collapse into single column layouts in mobile view.

Conclusion

In the next section, we will learn how to use CSS Flexbox to create responsive website designs. CSS Flexbox is ideal for working with a single dimension at a time: rows **or** columns.

While we will not cover CSS Grid at this time, it is well worth learning. CSS Grid is ideal when manipulating two-dimensions: rows **and** columns. It is used to create complex multi-column/multi-row layouts that are also responsive to smaller screens. To learn more about CSS Grid, see the following resources:

- CSS Grid Layout Module Level 1
- Grid: MDN
- Grid: W3 Schools

Flexbox Basics

CSS has long offered a number of *layout modes* for designing the layout of documents, text, tables, and positions. CSS Flexbox layout is a newer layout mode used to create responsive web page designs. It works in one-dimension at a time: rows **or** columns.

Introduction

CSS Flexbox includes the following **properties**:

- `flex-direction` sets the axis of the flex container. Options include:
  - `row`
  - `row-reverse`
  - `column`
  - `column-reverse`
- `flex-wrap` defines whether flex container is single or multi-line. Options include:
  - `nowrap` (single line)
  - `wrap` (multi-line)
  - `wrap-reverse`
- `flex-flow` is shorthand for: `flex-direction` and `flex-wrap`.
  - For example: `article { flex-flow: row wrap }` or `section { flex-flow: column nowrap}`.
- `justify-content` works to align flex items (items in a flex container) along the **main axis** defined in `flex-direction`. Options include:
  - `flex-start` : pack items at the line start
  - `flex-end` : pack items at the line end
  - `center` : pack items near the center
  - `space-between` : distribute items evenly across the line, placing first and last flex item near the each margin border
  - `space-around` : similar to above, but spaces first and last flex item a half space from each margin border
- `align-items` works on the **cross axis** of the container. Options include:
  - `flex-start` : flex items are flushed against the start of the container
  - `flex-end` : flex items are flushed against the end of the container
  - `baseline` : the baselines of flex items are aligned
  - `stretch` : flex items are stretched from start to end

- `align-content` is similarly to how `justify-content` works on the main axis but works on the **cross axis** when there is extra space. Takes effect when their are multiple lines of items, such as when `flex-wrap: wrap` is in effect. Options include:
    - `flex-start` : pack items at the container start
    - `flex-end` : pack items at the container end
    - `center` : pack items near the center of the container
    - `space-between` : distribute items evenly in the flex container, placing first and last flex item near the each margin border
    - `space-around` : similar to above, but spaces first and last flex item a half space from each margin border
    - `stretch` : flex items are stretched from start to end

Note how some properties above refer to the main axis or the cross axis.

- to use on main axis properties:
    - `justify-content`
- to use on cross axis properties:
    - `align-items`
    - `align-content`

By default, the main axis is the x-axis, and the cross axis is the y-axis.

Examples

To initiate a flex layout, we use the `display` property. We can use the `display` property in section elements such as `<body>` , `<article>` , `<section>` , `<nav>` , `<div>` , etc. These elements become **flex containers**. The **flex items** in these containers is thus any direct child element of a flex container. For example, to create a flex container for the `<nav>` element, we could use the following CSS declaration:

```
nav {
    display: flex;
    flex-flow: column wrap;
}
```

The elements in that `<nav>` section, such as an unordered list `<ul>` , become the container's flex items.

To create nested layouts, we can use `display` multiple times in a single web page. For example, if an `<article>` element contains several `<section>` elements, then the following would be used:

```
article {
    display: flex;
    flex-flow: column nowrap;
}

section {
    display: flex;
    flex-flow: row wrap;
    }
```

Basic Flex Example

In the following example, I make the `<section>` element a **flex container**. This element includes six `<p>` elements, which become the **flex items**. The `gap` property in the `section` rule adds a `10px` gap between between flex items. For the `<p>` element, I add borders and expand their dimensions to create boxes around these elements. I use a **structural pseudo class** `nth-child()` to alternate the colors of these boxes.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Flex Example</title>
        <style>
            html {
                font-size: 100%;
            }

            header {
                border-bottom: 10px dotted blue;
            }

            section {
                display: flex;
                gap: 10px;
                flex-wrap: wrap;
            }

            p {
                border: 1px solid black;
                font-size: 2rem;
                width: 8rem;
                height: 8rem;
                padding: 10rem;
            }

            p:nth-child(odd) {
                background-color: black;
                color: white;
            }

            p:nth-child(even) {
                background-color: yellow;
                color: black;
            }

            footer {
                border-top: 10px dotted blue;
            }
        </style>
    </head>

    <body>
        <header><h1>This is a title</h1></header>

        <section>
            <p>Box 1</p>
            <p>Box 2</p>
            <p>Box 3</p>
            <p>Box 4</p>
```

```
            <p>Box 5</p>
            <p>Box 6</p>
        </section>

        <footer><h2>This is a footer</h2></footer>
    </body>
</html>
```

Two Column Example

In the following example, I create a basic two-column layout. The HTML of the page contains a `<section>` element with two `<article>` elements. In the CSS, I make the `<section>` element the flex container. Since this element has the two `<article>` elements as child elements, the `<article>` elements become flex items. The items are wrapped (`flex-flow`) to make them responsive. And `justify-content` is used to center the items within the `<section>` container.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Flex Example: Two Column Layout</title>
        <style>
            html {
                font-size: 100%;
            }

            header {
                border-bottom: 2px black solid;
                text-align: center;
            }

            section {
                display: flex;
                gap: 50px;
                flex-flow: row wrap;
                justify-content: center;
            }

            article {
                margin: auto;
            }

            footer {

                border-top: 2px black solid;
                text-align: center;
            }
        </style>
    </head>

    <body>
        <header><h1>Main Title</h1></header>

        <section>
            <article>
                <h2>Article Title 1</h2>
                <p>This is some text.</p>
            </article>
            <article>
                <h2>Article Title 2</h2>
                <p>This is more text.</p>
            </article>
        </section>

        <footer><h2>This is a footer</h2></footer>
    </body>
</html>
```

Mixed Columns

In the following example, I add an additional row that isn't a flex container. The result is a top row that contains two columns and an additional single column row. I make several other changes to augment the look of the page. These include:

I add the following to the `body` selector to make the margins flush with the browser window:

```
body {
    max-width: 100%;
    margin: auto;
}
```

I create a `nav` section and make it a flex container. The result is a responsive navigation bar. The `<nav>` element includes an unordered list. I remove the bullets in this list to make the list's appearance more conventional for a navigation section.

```
nav ul {
    display: flex;
    flex-flow: row wrap;
    justify-content: space-around;
    background-color: black;
    color: white;
    padding: 10px;
    list-style-position: inside;
}

nav ul li {
    list-style-type: none;
}
```

Next comes the first of two `<section>` elements in the web document. I assign the first `<section>` element to a class called `main` to indicate that this is the main section, and to differentiate it from the second `<section>` element. In the CSS, I make the main section a flex container. I use `flex-flow: row wrap` to make it a responsive row, and I use `justify-content: center;` to center it on the main axis.

```
.main {
    display: flex;
    gap: 10px;
    flex-flow: row wrap;
    justify-content: center;
}
```

The remaining styles in the CSS control help keep the page consistent with the rest of the styling. Note the use of the `clamp()` function. This is used to set the minimum, preferred, and

maximum values of an element.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Flex Example: Mixed Column Layout</title>
        <style>
            html {
                font-size: 100%;
                font-family: monospace;
            }

            body {
                max-width: 100%;
                margin: auto;
                font-size: 1.5rem;
            }

            header {
                border-bottom: 2px black solid;
                text-align: center;
            }

            nav ul {
                display: flex;
                flex-flow: row wrap;
                justify-content: space-around;
                background-color: black;
                color: white;
                padding: 10px;
                list-style-position: inside;
            }

            nav ul li {
                list-style-type: none;
            }

            .main {
                display: flex;
                gap: 10px;
                flex-flow: row wrap;
                justify-content: center;
            }

            .main_article {
                margin: auto;
                max-width: clamp(320px, 40%, 1000px);
            }

            .secondary_article {
                margin: auto;
                max-width: clamp(320px, 40%, 1000px);
            }
```

```
        footer {
            border-top: 2px black solid;
            text-align: center;
        }
    </style>
</head>

<body>
    <header>
        <h1>Main Title</h1>
        <nav>
            <ul>
                <li>Home</li>
                <li>About</li>
                <li>Blog</li>
            </ul>
        </nav>
    </header>


    <section class="main">

        <article class="main_article">
            <h2>Article Title 1</h2>

            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
            do eiusmod tempor incididunt ut labore et dolore magna aliqua.
            Ut enim ad minim veniam, quis nostrud exercitation ullamco
            laboris nisi ut aliquip ex ea commodo consequat. Duis aute
            irure dolor in reprehenderit in voluptate velit esse cillum
            dolore eu fugiat nulla pariatur. Excepteur sint occaecat
            cupidatat non proident, sunt in culpa qui officia deserunt
            mollit anim id est laborum.</p>

        </article>

        <article class="main_article">

            <h2>Article Title 2</h2>

            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
            do eiusmod tempor incididunt ut labore et dolore magna aliqua.
            Ut enim ad minim veniam, quis nostrud exercitation ullamco
            laboris nisi ut aliquip ex ea commodo consequat. Duis aute
            irure dolor in reprehenderit in voluptate velit esse cillum
            dolore eu fugiat nulla pariatur. Excepteur sint occaecat
            cupidatat non proident, sunt in culpa qui officia deserunt.</p>

        </article>

    </section>

    <section>
```

```
            <article class="secondary_article">

                <h2>Article Title 3</h2>

                <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
                do eiusmod tempor incididunt ut labore et dolore magna aliqua.
                Ut enim ad minim veniam, quis nostrud exercitation ullamco
                laboris nisi ut aliquip ex ea commodo consequat. Duis aute
                irure dolor in reprehenderit in voluptate velit esse cillum
                dolore eu fugiat nulla pariatur. Excepteur sint occaecat
                cupidatat non proident, sunt in culpa qui officia deserunt.</p>

            </article>

        </section>

        <footer><h2>This is a footer</h2></footer>

    </body>
</html>
```

Conclusion

In this section we learned about the CSS Flexbox layout model. I first detailed the CSS properties that are available to create a Flex layout. I provided an example using basic boxes. Then I created a two-column example and a mixed column example, both of which are responsive to mobile views. These may also be adapted to create more complex layouts.

Note that the use of Flex aligns well with section elements, such as the `<article>`, `<section>`, `<main>`, elements. This is because, by definition, these elements are container elements.

Additional sources:

- CSS Flex Container
- Flexbox MDN

Document Metadata: JSON-LD and Schema.org

Introduction

Near the beginning of this work, we learned about the importance of structuring and adding metadata to the `<head>` section of our HTML documents. For example, the `<head>` section should contain a `<title>` element that accurately describes the content of the document.

We also learned about the `<meta>` element and how it can be used with the `name` attribute to describe the author and to provide a description and keywords of the web pages we create. Adding this metadata provides search engines with the information they need to return relevant results to their users.

As an example, the following HTML code snippet was introduced in the Document Structure and Metadata section of this work. The code snippet includes `<title>` and `<meta>` elements that describe a page about *Linux Systems Administration* authored by *C. Sean Burns*:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using the Linux OS for Systems Administration</title>
        <base href="https://www.example.org/">
        <link rel="stylesheet" href="css/style.css">
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="author" content="C. Sean Burns">
        <meta name="description" content="An introduction to Systems
Administration with Linux OS, including Debian and Ubuntu">
        <meta name="keywords" content="systems administration, linux, debian,
ubuntu">
        <style>
            html { font-family: monospace; }
        </style>
    </head>
    <body>
    </body>
</html>
```

In the HTML Semantic Elements and the HTML Grouping and Text-Level Semantics sections of this work, we were introduced to the ways that the HTML5 language created **semantic**

elements that help to add meaning to a document's structure. These elements are placed in the `<body>` section of a web page.

For example, the `<article>` element "represents a self-contained composition in a document, page, application, or site" (`<article>` : The Article Contents element). We might use the `<article>` element to capture a part of a web page that has an essay, blog post, news article, or like. Likewise, the `<nav>` element is a semantic element that "represents a section of a page whose purpose is to provide navigation links, either within the current document or to other documents" (`<nav>` : The Navigation Section element). Thus we use this to provide navigation on our websites.

Combined, via the elements we add to our `<head>` and `<body>` sections of our web pages, HTML5 provides tools to add metadata and semantic data to our web pages and thus help build **The Semantic Web**.

However, those elements are primarily aimed at adding semantic data about the document as a whole and its structure. In this last section of our work, we will explore how to add semantic data to our web pages that provide more information about the **content** of our those pages.

To do that, we will learn about the schema.org vocabulary. Schema.org is a **data model** that we will serialize as JSON-LD, or JSON Linked Data. Therefore, we will first learn how to structure data using JSON, which is a format for data exchange, primarily across the internet. And then we will learn how to serialize, or integrate, a schema.org vocabulary into JSON-LD.

By adding appropriate metadata in the `<head>` sections and semantic elements in the `<body>` sections of our web documents, and by adding JSON-LD metadata to our documents, we will make a contribution to the Semantic Web. The result will be websites that are more accessible to search engines, AI, and to our users.

In the next section, we will begin by learning how to create JSON objects. Then in the following section, we learn about the schema.org data model and how to use it. Finally, we will learn how to serialize the schema.org vocabulary as JSON-LD and add it to our web pages.

By the end of this chapter, you will be able to:

- Identify the purpose of JSON-LD and schema.org
- Use schema.org types and properties to describe content
- Embed structured data in a web page using JSON-LD

JSON Basics

Introduction

It may be too obvious to say, but computer applications use and exchange data in order to operate. For example, an OPAC (library catalog) contains bibliographic records (data) that allows users to search the data from a browser to locate information sources. An Apache web server stores configuration information in order to turn on, off, or fine tune the HTTP services it provides.

What may not be obvious is that data is stored in a variety of formats. Library science students may be familiar with XML (Extensible Markup Language) because it's used to serialize MARC data (see MARCXML) or DublinCore (DC) metadata. XML performs this service by storing data and metadata in a standardized format. CSV (comma separated values) files function as a *de facto* standard for storing tabular data. Data stored in CSV files are often used for data or statistical analysis, like in the R or Python programming languages. YAML files are used to store configuration files for various applications.

Oftentimes we might hear that data or metadata is **encoded** in a certain way, such as in MARC or DC, and then **encoded** again in XML, CSV, YAML, or similar formats. This isn't accurate, though. Encoding has other, more precise uses; for example, it's used to refer to **character encoding** standards that dictate how individual characters (like those on your keyboard) are represented in bytes on a computer. Major examples include UTF-8 or ASCII.

It's more precise to say that standards like MARC and DC function as **data models**. Data models are also known as **conceptual models** or as **schemas**. A data model dictates the kind of information that should be saved. For example, the DC schema for metadata includes data points like **title**, **subject**, **description**, and more.

When we write down the data in a file, we express the data using syntax. XML is an example of both a syntax (such as `<xml>`) and a format (such as file format like `index.xml`). Expressing data that conforms to a specific model (e.g., DC) using a syntax (e.g., XML) is called **serialization**. Thus, we say that DC data is serialized in XML syntax, or bibliographic data follows the MARC21 data model and is serialized as MARCXML.

Metadata

Metadata is data about data. When a librarian logs an entry for a book, they record metadata about the book that includes data points like: title, author, publisher, publication date, subject, and other descriptive elements. When we take a photo with our smartphone, our camera app records a name for the photo (e.g., `IMG_2025_04_02.jpg`), the date it was taken, the name and type of the device, the size of the photo, and oftentimes the geographical location of the photo.

Earlier in this work, we used the `<meta>` element to capture some of our web document's metadata in the `<head>` section. In this chapter, we will learn to add more specific metadata using the schema.org data model. Just as the DC data model dictates certain kinds of data points to collect (e.g., title, subject, description, etc.), the schema.org data model provides a vocabulary we can use to describe our web pages. Thus, using JSON syntax, we will serialize our schema.org data as JSON-LD, or JSON Linked Data.

The purpose of JSON-LD is to provide computers with data about the context and content of web pages. That is, the purpose is to provide *metadata*, or *data about data*, to computers, like search engines and AI, that allow them to build an *understanding* of our web pages. Major platforms like Google, Bing, and others use JSON-LD to understand web content specifically.

JSON

To understand how to serialize the schema.org data model into JSON-LD, we need to know how to use JSON. JSON (JavaScript Object Notation) is a format for data exchange, primarily across the internet. While its history is tied with the JavaScript programming language, it's not dependent on that language. Many programming languages have built-in or add-on libraries for working with data serialized in JSON.

JSON uses two data structures: objects and arrays (or lists). Per the JSON documentation:

> an *object* is an unordered set of name/value pairs. An object begins with a `{ left brace` and ends with `} right brace`. Each name is followed by `: colon` and the name/value pairs are separated by `, comma`.

And:

> An *array* is an ordered collection of values. An array begins with `[ left bracket` and ends with `] right bracket`. Values are separated by `, comma`.

Name/value pairs are separated by colons and both are **double quoted**:

```
"name": "value"
```

Values may take on specific data types that include:

- strings
- numbers
- JSON objects
- arrays
- Boolean ("true" or "false")
- null

A Simple JSON Object

Let's create a basic JSON object using the technical details just described. The example JSON object below begins to describe a person. The object begins with an opening curly brace and ends with a closing curly brace, and contains seven **name/value** pairs. All name/value pairs end with a comma except the last line. The last line of any JSON object **does not** end with a comma. The **field names** include `name`, `title`, `interests`, `worksFor`, `college`, `department`, and `program`. Each field name includes a corresponding value.

```
{
    "name": "C. Sean Burns",
    "title": "Associate Professor",
    "interests": "semantic web",
    "worksFor": "University of Kentucky",
    "college": "College of Communication and Information",
    "department": "School of Information Science",
    "program": "Information Communication Technology (ICT)"
}
```

A JSON Object with an Array

Some fields may take more than one value. For example, I happen to know that the person described in the simple JSON object above has more than one interest. Fortunately, we can use a JSON **array** to add more interests.

An array is assigned a name and begins and ends with square brackets. Each item in a JSON array ends with a comma, but like the last item in a JSON object, the last item in an array does not end with a comma. To see what this looks like, I convert the `interests` field to a array that lists several of this person's interests:

```
{
    "name": "C. Sean Burns",
    "title": "Associate Professor",
    "interests": [
        "semantic web",
        "scholarly communication",
        "open science",
        "open access",
        "information retrieval",
        "academic libraries"
    ],
    "worksFor": "University of Kentucky",
    "college": "College of Communication and Information",
    "department": "School of Information Science",
    "program": "Information Communication Technology (ICT)"
}
```

Nesting JSON Objects

As noted, JSON objects may take as a *value* other JSON objects. This is useful when a particular field name may be converted into an object itself. For example, in the above JSON objects, the `worksFor` field name can include other properties and thus may be extended into an **object**. Since the person described in the JSON object also has a work location, I can add that as a new object named as `location`. The result is the following JSON object that contains two nested JSON objects: `worksFor` and `location`. Note that since the person described in this object teaches in two programs, I converted the `program` field name into a list that includes both programs.

```
{
    "name": "C. Sean Burns",
    "title": "Associate Professor",
    "interests": [
        "semantic web",
        "scholarly communication",
        "open science",
        "open access",
        "information retrieval",
        "academic libraries"
    ],
    "worksFor": {
        "name": "University of Kentucky", // nested object describing organization
        "college": "College of Communication and Information",
        "department": "School of Information Science",
        "program": [
            "Information Communication Technology", // new array
            "Library Science"
        ]
    },
    "location": {
        "streetAddress": "326 Lucille Little Library", // nested object describing
location
        "addressLocality": "Lexington",
        "addressRegion": "Kentucky",
        "postalCode": "40506"
    }
}
```

Note: Comments (marked with `//`) in the above JSON object are shown here for explanation and are not allowed in actual JSON syntax.

JSON Linting

Just as we have validated our HTML5 and CSS3 code, we can also validate our JSON code. Validation programs examine the relevant code and check for syntax and other issues. When writing JSON syntax manually, use the JSONLint to check for syntax errors.

The popular `jq` JSON processor can help clean up and prettify JSON syntax. It is not a linter, but it can be used to examine, search, sort, and analyze JSON data.

Conclusion

In this section, we learned more about the concept of metadata, which generally relies on a **data model (or schema)** and serialization into a format, such as XML, CSV, and JSON. In the next section, we will begin to explore the full richness of the schema.org vocabulary and learn how to serialize that as JSON-LD.

Before we can use JSON-LD, we need to know how to understand and use JSON. We learned that JSON uses a `"name": "value"` syntax. Both `"name"` and `"value"` are double quoted. **Values** may include specific data types such as strings, numbers, JSON objects, arrays, Boolean, and null. Each `"name": "value"` item ends with a comma. The last `"name": "value"` item in a JSON object does not end in a comma. If a JSON object includes an array as a data type, the array is enclosed in square brackets `[ ]`. JSON objects nested in JSON objects are **named** and then the nested object is enclosed in curly brackets `{ }`.

As with HTML and CSS, remember that it's important to validate our JSON syntax. Therefore, we also learned about tools like the JSON Linter. The more complex our JSON objects become, the easier it is to introduce syntax errors into them. So be sure to validate your JSON.

When we begin working with schema.org, you will see how this structured, machine-readable format allows search engines and AI to *understand* what your website is about, even if no human ever reads it.

Schema.org and Structured Data

Introduction

Metadata plays a central role in most computer applications. There are different data models to structure metadata. The data model we will use to add metadata to our pages is called schema.org.

Data Models: An Overview

Some data models provide a basic structure, but others are more involved. Regardless, they all provide a sort of **controlled vocabulary**, which means they:

- reduce ambiguity by using approved terms
- promote interoperability between systems, platforms, or institutions
- allow for machine processing by enforcing structures and values.

For example, the Dublin Core data model provides a flat list of controlled terms. The main terms include:

- `contributor`
- `coverage`
- `creator`
- `date`
- `description`
- `format`
- `identifier`
- `language`
- `publisher`
- `relation`
- `rights`
- `source`
- `subject`
- `title`
- `type`

We can use this data model for all sorts of items or works. Here I use it to describe Toni Morrison's Pulitzer Prize winning novel, *Beloved*:

DC Term	Property
Creator	Toni Morrison
Title	Beloved
Publisher	Alfred A. Knopf Inc.
Language	English
Date	1987
Identifier	1-58060-120-0

Unlike Dublin Core, other controlled vocabulary data models are designed to be more interconnected. For example, MeSH (Medical Subject Headings) and LCSH (Library of Congress Subject Headings) are types of thesauri. Both are organized in a tree-like hierarchical structure; for example, MeSH terms are arranged from broad categories (e.g., **neoplasms**) to specific categories (e.g., **Neoplasm Metastasis**). Likewise, LCSH employs broader terms (BT, like **C (Computer Programming Language)**), related terms (RT, like **Objective-C (Computer Programming Language)**), and narrower terms (NT, like **Small-C (Computer Programming Language)**). MeSH is used in biomedical and health-related indexing, such as in PubMed. LCSH is often used by the Library of Congress and in academic libraries catalogs (e.g., UK's InfoKat).

Schema.org

Then there's **schema.org**, which was created by Google, Microsoft, Yahoo, and Yandex, for the purpose of describing web content for search engines. Unlike the prior data models, schema.org functions more like a taxonomy and ontology. As a taxonomy, schema.org is a kind of hierarchical, relational classification system, and as an ontology, schema.org stresses foundational components, such as concepts or classes, properties or attributes, relationships, and instances.

Like other data models, the schema.org vocabulary provides a method for adding structured, linked data. The data is *linked* because the vocabulary is interconnected via a hierarchical data model. This means that each type or property can point to or be reused across datasets, and it's this characteristic that creates a web of meaning that is readable by machines.

For example, the root data type in schema.org is `Thing`. The `Thing` type includes child data types such as `Action`, `Person`, `Place`, `Organization`, and more. These are all types of *Things* (or *classes*). The child data types include additional descendants; for example, the following are all examples of specific classes of an `Organization` thing:

- `Airline`,
- `EducationalOrganization`
- `PoliticalParty`,
- `LocalBusiness`, and more.

Digging deeper, if we focus on the `EducationalOrganization` type, we find that it may include other *Things*:

- `CollegeOrUniversity`
- `ElementarySchool`
- `HighSchool`, and so on.

Schema.org data types are *transitive* (if `a > b` and `b > c`, then `a > c`). For example, the *University of Kentucky* is an `instance` of a `CollegeOrUniversity` type. This itself is a subclass of an `EducationalOrganization`. We could go on: an `EducationalOrganization` type is a subclass of an `Organization` type. And finally, an `Organization` thing is a subclass of `Thing`. We might represent this as follows:

```
- Thing
    - Organization
        - EducationalOrganization
            - CollegeOrUniversity
                - University of Kentucky (instance)
```

All classes eventually descend back to the `Thing` type, just as in biology, all life on Earth is classified in a taxonomy with `Domain` holding the broadest rank.

Furthermore, all types have **properties**. A `Thing` type can have the following properties:

- `image`
- `name`
- `description`

And an `EducationalOrganization` Thing can have `alumni` as a property.

And each of those properties may have additional properties or take on values. For example, for `image`, we can provide a URL to an actual image. Or we may provide a `caption` for it.

However, just like `University of Kentucky` can be counted as an instance of `CollegeOrUniversity`, each `type` has its own set of **instances** in schema.org. To illustrate: since a `CollegeOrUniversity` thing is also an `EducationalOrganization` thing, a `CollegeOrUniversity` thing may also have the properties specific to `EducationalOrganization`, such as `alumni`. For instance, because `CollegeOrUniversity`

inherits from `Thing`, it can use general properties like `name`, `description`, and `url`, but also more specific ones like `alumni` that are directly inherited from `EducationalOrganization`.

That is, a `CollegeOrUniversity` thing may inherit properties of other types not in its direct lineage. Another example: a `CollegeOrUniversity` thing may also be a `CivicStructure` thing and a `Place` thing, even though neither of those are specific descendants of `EducationalOrganization`. In this way, specific `things` and `properties` can interconnect or `link` to each other, forming `linked data`. That is, it's this ability to belong to multiple classes, and to inherit properties from these classes, that enables schema.org to describe real-world complexity more naturally than rigid, single-hierarchy systems.

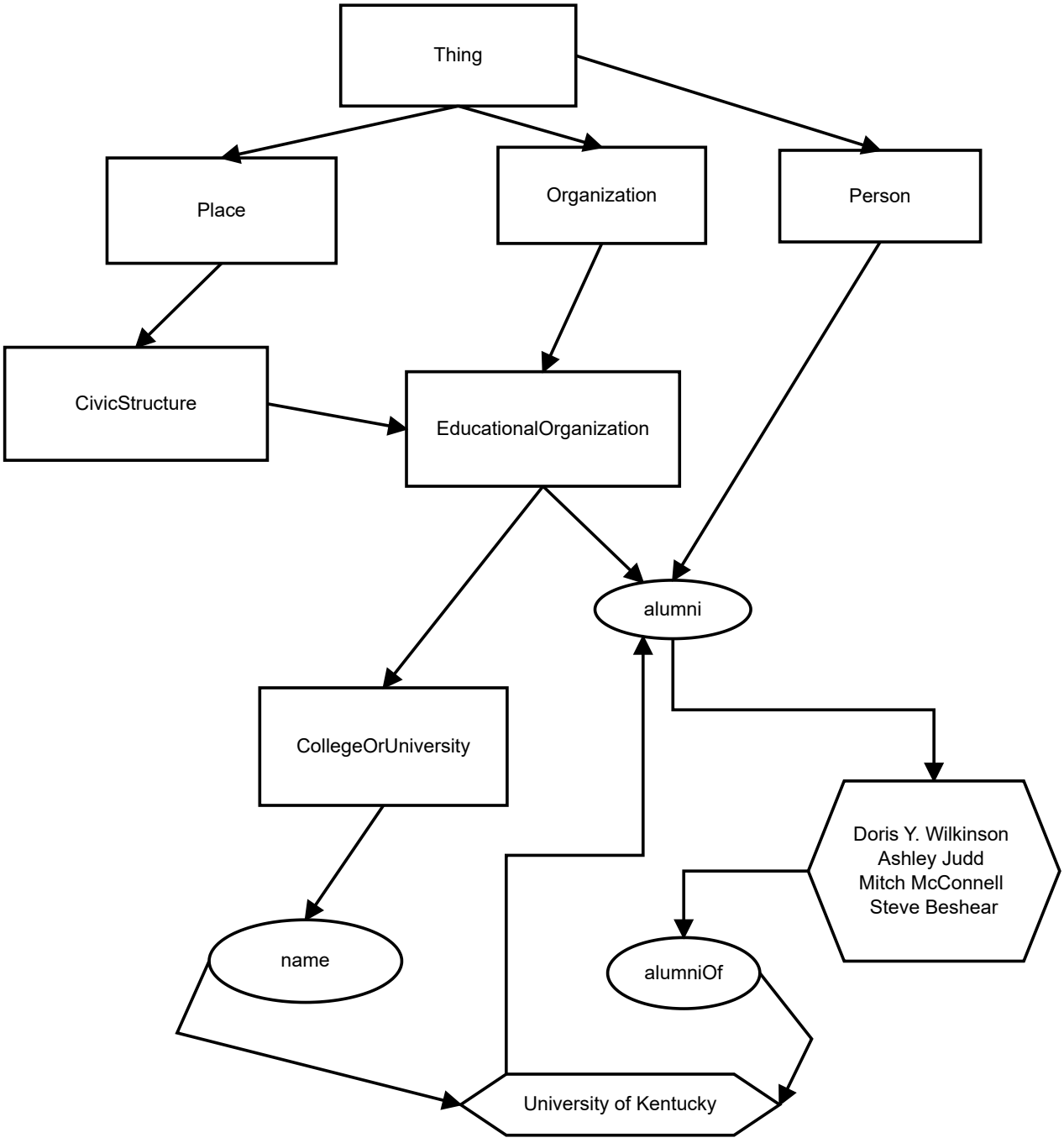


Fig. 2. Schema.org Example Map of University of Kentucky. *Types* are represented in square shapes. *Properties* are represented in oval shapes. *Instances* are represented in hexagonal shapes. Diagram created using Dia.

As you can see, a particular `instance` of some `Thing` may be a member of many classes, or be many types of `Things`. This is the same as you and me. For example, I am a professor, a parent, an offspring, etc. You might be a student and offspring. Thus, we both share at least one class, and inherit the properties of that class and its broader classes, like `Person`. By using this organizational model to describe the content of a web page, search engines can begin to *understand* that content and its context and the relationship among *Things* on the web.

To employ schema.org on your web pages requires some familiarity with the data model and what it offers. Therefore, begin reviewing the Full schema hierarchy for a complete listing of what is available.

Conclusion

In this section, we were introduced to the **schema.org** data model. We learned that schema.org is a hierarchical and extensible data model, with `Thing` as the root class and many descendant classes that inherit and extend its properties. By understanding this structure, we will be able to select the right types and properties that describe our web pages.

In the next section, we will use the schema.org data model to model the content of our web pages. Then we will serialize the models we create as JSON-LD.

To sum it up: Metadata serialized in JSON-LD is used by search engines, AI, and other services to *understand* the content expressed in HTML, the latter of which is used for human consumption. It accomplishes this through the schema.org vocabulary (although other data models exist for different contexts).

In the next section, we focus on the practical aspects of serializing `schema.org` as JSON-LD.

JSON-LD Syntax and Integration

Introduction

Now that we know how to structure data using JSON and use schema.org to conceptualize `Things`, we can begin to learn how to serialize schema.org into JSON-LD.

Step 1: The `<script>` tag

Our JSON-LD code will be placed within the `<head>` of our web pages. We will use the `<script>` HTML element with the `type` attribute. The `type` attribute will hold a value for the `ld+json` MIME type, which is `application/ld+json`. In the following snippet, I place this element just before the closing `</head>` tag. The JSON-LD metadata will be placed between the opening and closing `<script>` tags.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using the Linux OS for Systems Administration</title>
        <base href="https://www.example.org/">
        <link rel="stylesheet" href="css/style.css">
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="author" content="C. Sean Burns">
        <meta name="description" content="An introduction to Systems
Administration with Linux OS, including Debian and Ubuntu">
        <meta name="keywords" content="systems administration, linux, debian,
ubuntu">
        <style>
            html { font-family: monospace; }
        </style>
        <script type="application/ld+json">
        </script>
    </head>
    <body>
    </body>
</html>
```

Step 2: Creating a Data Model

In the last section, I showed how data models dictate the kind of metadata we can collect. I showed an initial example of how Dublin Core's fifteen main data fields can be used to capture specific metadata elements of a novel. That was a relatively easy example, and that's in large part because Dublin Core is a fairly flat and small schema. Knowing how to use schema.org to capture the important metadata elements is a bit more involved, and this is because schema.org is a big, general purpose taxonomy.

The first step in identifying the schema.org elements to use is to start with the content of the web page that you want to describe with schema.org. The next step is to identify the main topic(s) of that content. For example, is the main topic a person? If so, then we should start with the `Person` schema.org type. Or is the main topic an event, like a sporting event or a hackathon? Then we should start with the `SportsEvent` schema.org type or `Hackathon` schema.org type. Or are there multiple topics? In no way are you expected to memorize all the available `Types` that schema.org provides, however, identifying the needed `Types` does involve searching through `schema.org` to locate what's most relevant to your topic.

In the HTML snippet above, my made-up web page entails at least four topics:

- Systems Administration
- Linux
- Debian
- Ubuntu

Without knowing anything else, I can use these four topics to begin building a data model for my web page. However, I want to **reiterate** that in real-world practice, we build a JSON-LD data model based on the content and not just on what appears in the `<title>` and `<meta>` HTML tags. This is because JSON-LD is meant to capture details about the content in the web document and not just about the document overall.

For example, consider a recipe for Baked Kibbeh, a Lebanese dish made of lamb, bulgur wheat, onion, and spices, often eaten when the meat is raw and sometimes cooked. If we search the source code of the recipe's web page, we find that it uses JSON-LD. We can examine that JSON-LD by pasting the URL in the schema.org validator, and see that the entire recipe, including ingredients and recipe instructions, is described in the JSON-LD.

Content creators do not need to manually create their JSON-LD markup. The recipe site above likely uses JSON-LD generated by the Yoast SEO WordPress CMS plugin. Other CMS platforms can probably do similar things. The NY Times, the WSJ, and other major news platforms also embed JSON-LD and their JSON-LD are likely created through other automated mechanisms.

Step 3: Serializing schema.org as JSON-LD

When we create a JSON-LD data model for our web pages, we must use data extracted from the content on the web pages and no more. If our web page has an article on Linux Systems Administration, as suggested in the various `<meta>` elements in the above HTML snippet, then we must create a JSON-LD model based on that topic and on the content in that article **only**. If we add schema.org `Types`, `Properties`, and `Instances` to our JSON-LD that are not covered in the systems administration article, search engines will demote our website in their search results.

Here's a basic JSON-LD example that describes the Debian GNU/Linux distribution, one of the topics of my made-up webpage. Note that a JSON-LD object begins with an `@context` statement and a link to the schema.org site. Then, the `@type` is declared:

```
<script type="application/ld+json">
{
    "@context": "https://schema.org",
    "@type": "SoftwareApplication",
    "name": "Debian GNU/Linux",
    "description": "The Universal Operating System",
    "operatingSystem": "Debian",
    "softwareVersion": "12.10",
    "applicationCategory": "Operating System",
    "applicationSubCategory": "Linux Distribution",
    "memoryRequirements": "780MB",
    "storageRequirements": "1160MB",
    "url": "https://www.debian.org/",
    "downloadUrl": "https://www.debian.org/distrib",
    "releaseNotes": "https://www.debian.org/News/2025/20250315",
    "datePublished": "2025-03-15"
}
</script>
```

The above example is based on `SoftwareApplication` type. This is declared in the second line with `"@type": "SoftwareApplication"`. This type entails a range of properties but also inherits properties from parent/ancestor types. That is, since `SoftwareApplication` is a descendant/subclass of the `CreativeWork` type, which itself is a subclass of `Thing` type, then `SoftwareApplication` can use properties from itself but also from `CreativeWork` and `Thing`. Therefore, in the JSON-LD code above, I use properties from `SoftwareApplication` but also from `CreativeWork` and `Thing`, as detailed at the schema.org link above and in the list below:

```
- Thing (TYPE), properties of Thing:
    - name
    - description
    - url
        - CreativeWork (TYPE), property of CreativeWork:
            - datePublished
                - SoftwareApplication (TYPE), properties of SoftwareApplication:
                    - applicationCategory
                    - applicationSubCategory
                    - downloadUrl
                    - memoryRequirements
                    - operatingSystem
                    - releaseNotes
                    - softwareVersion
                    - storageRequirements
```

The above JSON-LD is a good start, but we can capture more metadata about the Debian operating system. For example, the Debian OS is overseen by *The Debian Project*, which is a legal entity and the creator of the OS. We can modify our JSON-LD code to include that metadata. Specifically, we will use the `creator` property, which accepts either the `Organization` or `Person` properties as values (or instances). Since both instances are schema.org `Types`, we can create a nested JSON object: To capture more detail, we can list the main features of Debian using the `featureList` property of the `SoftwareApplication` type. I use a JSON list (using square brackets) to list the system's features. Remember that square brackets `[]` indicate a list of values (an array), while curly braces `{}` represent a nested object. If you need a refresher on how arrays and nested objects work in JSON, see the earlier section on JSON Basics.

```
<script type="application/ld+json">
{
    "@context": "https://schema.org",
    "@type": "SoftwareApplication",
    "name": "Debian GNU/Linux",
    "description": "The Universal Operating System",
    "operatingSystem": "Debian",
    "softwareVersion": "12.10",
    "applicationCategory": "Operating System",
    "applicationSubCategory": "Linux Distribution",
    "memoryRequirements": "780MB",
    "storageRequirements": "1160MB",
    "url": "https://www.debian.org/",
    "downloadUrl": "https://www.debian.org/distrib",
    "releaseNotes": "https://www.debian.org/News/2025/20250315",
    "datePublished": "2025-03-15",
    "license": "https://www.debian.org/legal/licenses/",
    "creator": {
        "@type": "Organization",
        "name": "Debian Project",
        "url": "https://www.debian.org/intro/about"
    },
    "featureList": [
        "free software",
        "stable",
        "secure",
        "extensive hardware support",
        "flexible installer",
        "smooth upgrades",
        "community-developed",
        "long-term support"
    ]
}
```

Finally, we're missing a subtle addition to the JSON-LD. My made-up web page is about using Debian for systems administration. It's not about **Debian**, per se. Thus, the above JSON-LD might be appropriate on the Debian homepage, but my web page talks about using Debian rather than being about Debian. Understanding this requires acquiring a judgment of thing's aboutness. In any case, to capture this distinction, I add the WebPage type to the JSON-LD, along with `name` and `description` properties, and make the rest of the JSON nested in that type. I also name the nested JSON-LD object as `about`:

```
<script type="application/ld+json">
{
    "@context": "https://schema.org",
    "@type": "WebPage",
    "name": "Introduction to Linux Systems Administration with Debian",
    "description": "An educational web page covering systems administration with
Linux, focusing on Debian GNU/Linux.",
    "about": {
        "@type": "SoftwareApplication",
        "name": "Debian GNU/Linux",
        "description": "The Universal Operating System",
        "operatingSystem": "Debian",
        "softwareVersion": "12.10",
        "applicationCategory": "Operating System",
        "applicationSubCategory": "Linux Distribution",
        "memoryRequirements": "780MB",
        "storageRequirements": "1160MB",
        "url": "https://www.debian.org/",
        "downloadUrl": "https://www.debian.org/distrib",
        "releaseNotes": "https://www.debian.org/News/2025/20250315",
        "datePublished": "2025-03-15",
        "license": "https://www.debian.org/legal/licenses/",
        "creator": {
            "@type": "Organization",
            "name": "Debian Project",
            "url": "https://www.debian.org/intro/about"
        },
        "featureList": [
            "free software",
            "stable",
            "secure",
            "extensive hardware support",
            "flexible installer",
            "smooth upgrades",
            "community-developed",
            "long-term support"
        ]
    }
}
</script>
```

Step 4: Validating Our JSON-LD

It's quite easy to make mistakes when manually writing JSON-LD code. Therefore, be sure to test your JSON-LD using the Schema Markup Validator. The validator will test both the syntax of your JSON-LD but also show if any schema.org types or properties are missing.

Also, the JSON-LD Playground is not a validator, but it's useful for visualizing your code.

You can also use the Rich Results Test tool. From the site, “Rich results are experiences on Google surfaces, such as Search, that go beyond the standard blue link. Rich results can also include carousels, images, or other non-textual elements.” In other words, Rich Results are those results that feature prominently at the top of a Google search and generally include images and other descriptive content about a search result. For this to work, include images on your site and describe them in your JSON-LD. Also, these types of results are based on a subset of structured data. See: Structured data markup that Google Search supports.

Step 5: Inserting JSON-LD in Your HTML Document

Now that I've completed my JSON-LD object that describes one aspect of the topic of my Linux Systems Administration page, I can add it to my web document. To do that, I insert the JSON-LD object into the `<head>` section. The following snippet illustrates this:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using the Linux OS for Systems Administration</title>
        <base href="https://www.example.org/">
        <link rel="stylesheet" href="css/style.css">
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="author" content="C. Sean Burns">
        <meta name="description" content="An introduction to Systems
Administration with Linux OS, including Debian and Ubuntu">
        <meta name="keywords" content="systems administration, linux, debian,
ubuntu">
        <style>
            html { font-family: monospace; }
        </style>
<script type="application/ld+json">
{
    "@context": "https://schema.org",
    "@type": "WebPage",
    "name": "Introduction to Linux Systems Administration with Debian",
    "description": "An educational web page covering systems administration with
Linux, focusing on Debian GNU/Linux.",
    "about": {
        "@type": "SoftwareApplication",
        "name": "Debian GNU/Linux",
        "description": "The Universal Operating System",
        "operatingSystem": "Debian",
        "softwareVersion": "12.10",
        "applicationCategory": "Operating System",
        "applicationSubCategory": "Linux Distribution",
        "memoryRequirements": "780MB",
        "storageRequirements": "1160MB",
        "url": "https://www.debian.org/",
        "downloadUrl": "https://www.debian.org/distrib",
        "releaseNotes": "https://www.debian.org/News/2025/20250315",
        "datePublished": "2025-03-15",
        "license": "https://www.debian.org/legal/licenses/",
        "creator": {
            "@type": "Organization",
            "name": "Debian Project",
            "url": "https://www.debian.org/intro/about"
        },
        "featureList": [
            "free software",
            "stable",
            "secure",
            "extensive hardware support",
            "flexible installer",
            "smooth upgrades",
            "community-developed",
            "long-term support"
        ]
```

```
    }
}

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

If I have additional topics, I could add additional types to my JSON-LD. Generally, it's good to use JSON-LD that's fairly descriptive of the content on the page.

Note that JSON-LD isn't completely reusable across a site, since it's meant to mirror the content of a specific page. Therefore, each webpage on a website should include unique JSON-LD.

Conclusion

In this section, we explored how to describe web content using schema.org and JSON-LD. We learned how to model page content with appropriate types and properties, use JSON syntax to serialize metadata, validate the results, and add your JSON-LD to your web page.

Conclusion

The purpose of this textbook is to serve as an introduction to semantic web development. It does this by focusing on the fundamentals: HTML5, CSS3, and JSON-LD. Complete, usable, accessible, and semantically rich sites can be created with just these three technologies. No JavaScript required.

Some call this The rule of least power. The idea is to use the least powerful, most descriptive programming language before using more procedural programming languages. This is incredibly effective for web development for at least several reasons. First, the web is built first on HTML as the base technology and is backwards compatible to prior HTML versions. This means that if all the more complicated technologies (like JavaScript, etc) fail, HTML should still load. CSS probably will, too. Second, HTML, and even CSS, are fast because the basics of them are simple. They are incredibly focused on the document.

Last, HTML is highly tolerant. This means we can make a lot of mistakes in HTML, and web browsers will adjust. They won't even **crash**! If we make a mistake in our JavaScript code, the JavaScript will fail. Maybe that means the web page won't load, or more severely, the browser won't grind to a halt and crash.

I should stress the importance of the document more. Focusing on HTML allows us to focus on a document's structure and content, both of which inform each other and make web pages accessible. They also serve as the basis for CSS and JavaScript. These other, more complicated technologies require access to the DOM, or the Document Object Model, and as a markup language, HTML's focus on the document underpins the DOM structure used by both CSS and JavaScript. Thus, creating sound HTML makes it easier to create sound CSS and sound JavaScript.

Using HTML to focus on structure and content also helps with good thinking. That is, since HTML provides many semantic, section elements, it can help us think clearly about the meaning and structure of our documents. The structure of a thing is like a model of a thing. Structuring, modeling, framing: these are not just ways of organizing something like a document. They are also ways of thinking about content, logic, reasoning, and evidence.

While this book does not cover other web technologies like PHP, JavaScript, JavaScript frameworks like React, or relational databases like MySQL, the skills you've developed here form the foundation for all future web development. Plus, those other technologies are less for creating good web pages and more for building web applications. Use them wisely.

As you practice and acquire proficiency with the basics covered here, I invite you to keep experimenting. There are a lot of basics we haven't covered. For instance, since CSS is *rule-based*, there is a lot more to learn about writing conditional statements that initiate sets of

rules depending on the context, the device, and the browser. There are other so-called at-rules to learn, too. Furthermore, there is more to learn about CSS layouts, like CSS Grid, which allows for more complicated layout designs than CSS Flexbox.

Thanks, have fun, learn a lot, and study deeply.

Final Project: Implementing a Semantic Web Site

The following is a sample final project assignment students should be able to do upon completion of this book.

Overview

Your final project is to design and build a small, semantically structured website that reflects your understanding of HTML5, CSS3, Flexbox layout, JSON-LD, and web accessibility principles. This project gives you the opportunity to integrate the core concepts we've explored throughout this work into a cohesive and standards-compliant site.

Instructions and Project Requirements

Please follow these steps in preparing your assignment. Your final website must include the following:

Step 1: Site Structure

- A total of **three, distinct HTML pages**.
    - Each page must have its own substantial content.
- Each page must include consistent **navigational links** connecting all three pages.
- Use of **semantic HTML5 elements** appropriately; e.g., `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<footer>`, etc.
- All pages must link to the **same external CSS stylesheet**.
- Use **Flexbox only** for layout (no float-based or media query layouts accepted. Some Grid may be accepted but the overall layout should be dictated by Flexbox).

Step 2: Semantic Web Component

- Each HTML page must include **JSON-LD** in a `<script type="application/ld+json">` block in the `<head>`.
- The JSON-LD must be **unique to each page** and reflect that page's **actual content**.

Step 3: Testing and Validation

- **Validate** your HTML5 and CSS3 using the HTML5 Validator and CSS3 Validator.
  - You must have no **errors** in your code (you should fix warnings but this is optional).
- **Validate** your JSON-LD using the schema.org validator for all three pages.
- **Review the accessibility** of your color scheme using WAVE (web accessibility evaluation tool).
- **Turn on accessibility features** (like screen readers, high contrast mode, etc) on your **laptop or smartphone** and review your own site from that perspective.
  - This step builds on our earlier activity where you explored accessibility modes of various websites.

Step 4: Reflection Statement

Include a brief (400-600 words) reflection in which you:

- Discuss what you learned from the project.
- Reflect on challenges you encountered and how you addressed them.
- Include a focused section on **accessibility**:
  - What did you discover when testing your site in accessibility mode?
  - How might you improve the accessibility of your site further?

Step 5: Submitting Your Assignment Checklist

- A **single** document (PDF or Word) containing a:
  - Link to your website.
  - Link to your GitHub repository for your website.
  - WAVE review summary or screenshot.
  - HTML5/CSS3 validator conformation (screenshots or links).
  - JSON-LD validator confirmation (screenshots or links).
  - Reflection statement.