WEB DEVELOPMENT

FULL COURSE

2024

**SECTION ONE**

**INTRODUCTION**

Before you can understand how to program the web, you need an understanding of the web itself. These concepts provide an understanding of the ecosystem in which you will be working and will enable you to communicate effectively with other developers about your work.

This section contains topics that you will learn in this lesson.

* The internet.
* Data packets
* Web page, web server, web browser and search engine.
* Client and Server
* DNS servers
* IP address

**THE INTERNET**

Imagine a giant spiderweb, but instead of catching flies, it connects computers and devices all over the world. That's kind of like the internet! It's a massive network that uses cables, satellites, and even wireless signals to link everything together.

Here's a breakdown of the key points:

* **Network of devices:** The internet connects billions of computers, phones, tablets, and other gadgets.
* **Global reach:** It stretches across the entire planet, allowing information to flow freely between continents and countries.
* **Communication and information:** This network is the foundation for many things we do online, like browsing websites, sending emails, or chatting with friends.

Think of it as a giant information highway. You can use it to:

* **Access information:** Find almost anything you can imagine on the web, from news articles and educational resources to funny cat videos.
* **Connect with people:** Chat with friends and family, meet new people, or join online communities.
* **Shop and bank:** Buy products, pay bills, and manage your finances securely.
* **Do almost anything!:** The possibilities are endless!

The internet is a complex system, but that's the basic idea. It's a powerful tool that has revolutionized the way we live, work, and learn.

**DATA PACKETS**

Data packets are the unsung heroes of the internet, acting like tiny delivery trucks on the information highway. Here's how they work:

**Imagine sending a large box across town:**

* **Too Big, Too Slow:** If you tried sending the entire box at once, it would be slow and inefficient.

**Data Packets Break it Down:**

* **Chunked Up:** Data packets take large files (emails, videos) and break them down into smaller, more manageable pieces.
* **Like Packing Boxes:** Think of it like packing the contents of the big box into several smaller boxes.

**Each Packet has a Purpose:**

* **Two Parts:** Each data packet is like a mini-package with two sections:
  + **Payload:** This is the actual data you're sending, like the contents of your email or a portion of the video. (Think of it as what's inside the smaller boxes).
  + **Header:** This part acts like an address label. It contains information about the sender, receiver, and how to reassemble the data (like the recipient's address and instructions for fragile items).

**Taking Different Routes:**

* **Network Highways:** Data packets don't necessarily travel together. They might take different paths on the network, depending on traffic and efficiency. (Imagine taking different side roads to avoid congestion).

**Putting it Back Together:**

* **The Receiving End:** Once the packets reach their destination, the receiving device reassembles them based on the information in the headers. (Just like unpacking all the smaller boxes to get the original contents).

**The Power of Packets:**

* **Efficient Flow:** Data packets make information transfer on the internet much faster and more reliable. If a packet gets lost, only that specific piece needs to be resent, not the entire file. (Think of how much easier it is to replace one lost box than the whole shipment).

So, next time you browse the web, watch a video, or send an email, remember the tiny data packets working tirelessly behind the scenes to deliver the information you need!

**WEB PAGE, WEB SERVER, WEB BROWSER AND SEARCH ENGINE.**

These four terms are all essential parts of navigating the web, but they each play different roles:

**Web Page:**

* Think of a web page as a single document you see online. It's like a specific location on a website, containing text, images, videos, and interactive elements.
* You can access a web page through a web address (URL).
* Example: This very page you're reading right now is a web page.

**Web Server:**

* Imagine a web server as a giant digital warehouse. It stores all the web pages and files that make up a website.
* When you request a web page using your browser, the web server finds the corresponding files and sends them to your device.
* Web servers are powerful computers that are always connected to the internet.

**Web Browser:**

* This is the software you use to access web pages. It's like a special app that can interpret the information from a web server and display it in a user-friendly format on your screen.
* Popular web browsers include Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge.
* Browsers can also help you navigate between web pages and interact with them using features like links, buttons, and forms.

**Search Engine:**

* A search engine is like a giant library catalog for the internet. It helps you find specific information online by indexing websites and keeping track of their content.
* When you enter a search query, the search engine searches its index and provides you with a list of relevant web pages.
* Popular search engines include Google, Bing, DuckDuckGo, and Yahoo!.

Here's an analogy to tie it all together:

* Imagine a restaurant website. The **web pages** are the individual menus, photos, and online ordering sections. The **web server** is the restaurant's kitchen where all the recipe information and ingredients are stored. You use your **web browser** (like a phone app) to access the menus and order food. Finally, a **search engine** would be like a food review website that helps you find restaurants based on your preferences.

**CLIENT AND SERVER**

In the world of computers and the internet, client and server refer to specific ways devices communicate and share resources. Here's a breakdown:

**Client:**

* Think of a client as a program or device that requests information or services from another program, the server.
* Clients are often user-facing interfaces, like web browsers, email applications, or mobile apps.
* When you use a web browser to visit a website, the browser acts as the client, sending a request to the server for the web page you want to see.

**Server:**

* Imagine a server as a powerful computer that stores data and responds to requests from clients.
* Servers are always connected to the internet and run specialized software to manage incoming requests and deliver the desired information or service.
* When a client (like your web browser) makes a request, the server locates the information (the web page) and sends it back to the client for display.

**The Client-Server Dance:**

* This communication between client and server follows a request-response model:
  1. **Request:** The client initiates the conversation by sending a request to the server. This request specifies what information or service is needed.
  2. **Response:** The server receives the request, processes it, and sends back a response to the client. The response might be the data or service requested, or an error message if there's a problem.

**Benefits of Client-Server:**

* This architecture offers several advantages:
  + **Efficiency:** Servers are centralized and powerful, allowing them to handle multiple client requests simultaneously.
  + **Scalability:** You can easily add more clients or servers to meet growing demands.
  + **Security:** Servers can be secured to restrict access to sensitive data, while client devices typically don't store large amounts of data.

**Examples of Client-Server in Action:**

* **Web browsing:** Your web browser (client) requests web pages from a web server (server).
* **Email:** Your email client (client) fetches emails from a mail server (server).
* **Online games:** Your game client (client) connects to a game server (server) to play with others.

So, the next time you browse the web, check your email, or play an online game, remember the client-server dance happening behind the scenes to make it all work!

**DNS SERVERS**

DNS servers, short for Domain Name System servers, are the unsung heroes of the internet. They act like a giant phonebook for the web, translating the user-friendly website names we use (like google.com) into the numerical IP addresses that computers actually understand.

Here's how it works:

1. **Domain Name vs. IP Address:** Imagine you want to visit your friend's house. You know their name (like "google.com"), but your GPS needs an address (like an IP address) to find the location.
2. **You Ask, DNS Answers:** When you type a website name into your browser, your computer contacts a DNS server. It's like asking the phonebook for your friend's address.
3. **Behind the Scenes:** The DNS server receives your request and searches its database for the website name. This database contains millions of entries, mapping domain names to their corresponding IP addresses.
4. **The Answer is Back:** Once the DNS server finds the IP address, it sends it back to your computer.
5. **Connecting You:** Your computer now has the information it needs (the IP address) and can connect to the website's server.

**Benefits of DNS Servers:**

* **Memorize Friendly Names:** We can remember names like "google.com" much easier than complex IP addresses like "172.217.160.66" (which is actually a Google IP).
* **Dynamic Updates:** Websites can change servers, but the DNS system updates automatically to reflect the new location.
* **Efficiency:** DNS servers distribute the load of translating domain names, making web browsing faster.

**Types of DNS Servers:**

There are different levels of DNS servers working together to find the right IP address:

* **Recursive Resolver:** This is the DNS server you typically interact with first. It might be provided by your internet service provider (ISP).
* **Root Nameservers:** These servers point to the top-level domains (like ".com" or ".org").
* **TLD Nameservers:** These servers manage specific top-level domains.
* **Authoritative Nameservers:** These are the servers for a specific domain name and hold the final say on its IP address.

So next time you effortlessly type a website name and land on your desired page, remember the invisible army of DNS servers working tirelessly in the background to make your internet experience smooth and user-friendly!

**IP ADDRESSES**

An IP address, short for Internet Protocol address, is like a unique digital fingerprint assigned to each device connected to a network that uses the Internet Protocol (IP) for communication. Just like a street address helps mail reach your house, an IP address helps data packets find the right device on the vast network.

Here's a breakdown of what IP addresses are and how they work:

* **Identification:** An IP address is a string of numbers (separated by periods) that identifies a specific device. It's like an identification tag for your device on the internet.
* **Two Versions:** There are two main types of IP addresses in use today:
  + **IPv4:** This is the older version, consisting of four numbers between 0 and 255, separated by dots (e.g., 192.168.1.1). Due to the limited number of combinations, IPv4 addresses are becoming scarce.
  + **IPv6:** This is the newer version, designed to address the limitations of IPv4. IPv6 addresses are much longer and more complex than IPv4 addresses (e.g., 2001:0db8:85a3:0000:0000:8a2e:0370:7334).
* **Not Random:** IP addresses aren't randomly assigned. Internet authorities manage them and distribute them to local internet service providers (ISPs) who then assign them to devices on their network.
* **Public vs. Private:** There are two main categories of IP addresses:
  + **Public IP Addresses:** These are unique addresses assigned to devices directly connected to the internet. Think of it as your house's address on a public street.
  + **Private IP Addresses:** These are used for devices on a local network (like your home Wi-Fi). They are not directly routable on the public internet, but they allow devices on the same network to communicate with each other. Imagine these as addresses within a private apartment complex.

**What IP Addresses Do:**

IP addresses play a crucial role in enabling communication on the internet:

* **Routing Data:** When you request a website on your browser, your device sends out a data packet with your IP address as the sender and the website's IP address as the receiver. This allows routers and other network devices to route the data packets to the correct destination.
* **Finding Devices:** With an IP address, other devices on the network can locate and communicate with your device. This is essential for online activities like video calls, online gaming, or file sharing.

**Understanding IP addresses is not essential for everyday internet use, but it's a fundamental concept behind how devices connect and communicate on the vast network.**

**SECTION 2**

**INSTALLATION**

* Installing a code editor.

**INSTALLING A CODE EDITOR**

For this lesson visual studio code is recommended as it is easier to set up and has a lot of tools for making programming easier.

Installing a code editor like Visual Studio Code is a straightforward process. Here's how to do it for Windows or Mac its basically the same thing:

**Downloading Visual Studio Code:**

1. Head over to the official Visual Studio Code download page: <https://code.visualstudio.com/download>
2. You'll see different buttons for different. Click on the one for your OS to download the installer file.

**Running the Installer:**

1. Once the download is complete, locate the downloaded file. It will likely be named something like "VSCodeUserSetup-x64.exe" (the exact name may vary slightly depending on the version).
2. Double-click the downloaded file to launch the installer.

**Installation Process:**

1. The installer will walk you through the setup process. You'll typically see options like:
   * **License Agreement:** You'll need to agree to the license terms to proceed.
   * **Start Menu Folder:** You can choose to create a Start Menu folder for VS Code or skip this step.
   * **Desktop Shortcut:** You can decide if you want a shortcut icon for VS Code on your desktop.
   * **Additional Components:** You can choose to install optional components during setup, but the defaults are usually fine for beginners.
2. Once you've reviewed the options, click the "Install" button. The installer will unpack the necessary files and configure VS Code on your system.

**Launching VS Code:**

1. After successful installation, the option to launch VS Code might be presented. You can also find it from your Start Menu or by double-clicking the VS Code application file in the installation directory.

**Congratulations!** You should now have Visual Studio Code up and running on your machine.

**Additional Tips:**

* Make sure you have an internet connection during installation, as VS Code might download some additional components during the process.
* You can customize the look and feel of VS Code with themes and extensions available in the VS Code Marketplace.
* There are plenty of resources and tutorials available online to help you learn more about Visual Studio Code and its features.

**COMMAND LINE BASICS**

That blank screen or window with a prompt and blinking cursor is the command line interface (CLI), where you’re able to enter commands that your computer will run for you. While there’s no need for you to reenact the scene above, working with the command line is a critical skill for you to learn as a developer. The command line is like our base of operations, from which we can launch other programs and interact with them. It has a syntax of its own to learn, but since you’ll be entering the same commands dozens of times, you’ll quickly pick up the commands you need most.

In this introductory lesson to the command line, you’ll learn how to navigate around your computer and how to manipulate files and directories (also known as folders) directly from the comfort of the command line. You’ll soon see that this isn’t as difficult as you may think. The commands you will learn in this lesson are very straightforward. So don’t let the prospect of using the command line for the first time intimidate you.

**USING THE COMMAND LINE**

The way you open the command line (also sometimes called terminal or console) depends on your operating system:

**Windows:**

There are a few ways to open the command line in Windows:

* **Search bar:** Click on the Start menu or Windows icon and type "cmd" in the search bar. Press Enter to launch the Command Prompt.
* **Run program:** Press the Windows key + R on your keyboard to open the Run program. Type "cmd" and press Enter.
* **Power User Menu:** Press the Windows key + X or right-click on the Start menu icon. Select "Command Prompt" (or "Terminal" in Windows 11) from the menu.

**macOS/Linux:**

Most macOS and Linux systems come pre-installed with a terminal application. Here's how to access it:

* **Spotlight Search:** Open Spotlight Search (usually by pressing Command + Space) and type "Terminal". Click on the Terminal application icon to launch it.
* **Applications Folder:** Go to the Applications folder and navigate to the "Utilities" folder. You should find the Terminal application there.

Once you've opened the command line, you can type commands and press Enter to execute them.

**HTML and CSS**

HTML and CSS are two languages that work together to create everything that you see when you look at something on the internet. HTML is the raw data that a webpage is built out of. All the text, links, cards, lists, and buttons are created in HTML. CSS is what adds style to those plain elements. HTML puts information on a webpage, and CSS positions that information, gives it color, changes the font, and makes it look great!

Many great resources out there keep referring to HTML and CSS as programming languages, but if you want to get technical, labelling them as such is not quite accurate, because they are only concerned with presenting information. They are not used to program any logic. JavaScript, which you will be learning in the next section, is a programming language because it’s used to make webpages do things. Yet, there is quite a lot you can do with just HTML and CSS, and you will definitely need them both.

**HTML ELEMENTS AND TAGS**

Almost all elements on an HTML page are just pieces of content wrapped in opening and closing HTML tags.

Opening tags tell the browser this is the start of an HTML element. They are comprised of a keyword enclosed in angle brackets <>. For example, an opening paragraph tag looks like this: <p>.

Closing tags tell the browser where an element ends. They are almost the same as opening tags; the only difference is that they have a forward slash before the keyword. For example, a closing paragraph tag looks like this: </p>.

A full paragraph element looks like this:

<p>some text content</p>

Let’s break this down:

* <p> is the opening tag.
* some text content represents content wrapped within the opening and closing tags.
* </p> is the closing tag.

You can think of elements as containers for content. The opening and closing tags tell the browser what content the element contains. The browser can then use that information to determine how it should interpret and format the content.

HTML has a vast list of predefined tags that you can use to create all kinds of different elements. It is important to use the correct tags for content. Using the correct tags can have a big impact on two aspects of your sites: how they are ranked in search engines; and how accessible they are to users who rely on assistive technologies, like screen readers, to use the internet.

Using the correct elements for content is called semantic HTML. We will explore this in much more depth later on in the curriculum.

**Void Elements**

Some HTML elements do not have a closing tag. These elements just have a single tag, like: <br> or <img>. They are known as void elements because they are void of any content, there is nothing inside of them. No closing tag means they can’t wrap content like other tags do.

You might also see these referred to as self closing tags. But those are just void elements with a forward slash(/) at the end like: <br /> or <img />. You’re likely to see self-closing tags used often for historical reasons. Browsers will be able to render them just fine, but the latest version of the HTML specification discourages their use and considers them invalid.

**HTML BOILERPLATE**

All HTML documents have the same basic structure or boilerplate that needs to be in place before anything useful can be done. In this lesson, we will explore the different parts of this boilerplate and see how it all fits together.

To demonstrate an HTML boilerplate, we first need an HTML file to work with.

Create a new folder on your computer and name it html-boilerplate. Within that folder create a new file and name it index.html.

You’re probably already familiar with a lot of different types of files, for example doc, pdf, and image files.

To let the computer know we want to create an HTML file, we need to append the filename with the .html extension as we have done when creating the index.html file.

It is worth noting that we named our HTML file index. We should always name the HTML file that will contain the homepage of our websites index.html. This is because web servers will by default look for an index.html page when users land on our websites – and not having one will cause big problems.

[The DOCTYPE](https://www.theodinproject.com/lessons/foundations-html-boilerplate#the-doctype)

Every HTML page starts with a doctype declaration. The doctype’s purpose is to tell the browser what version of HTML it should use to render the document. The latest version of HTML is HTML5, and the doctype for that version is <!DOCTYPE html>.

The doctypes for older versions of HTML were a bit more complicated. For example, this is the doctype declaration for HTML4:

However, we probably won’t ever want to be using an older version of HTML, and so we’ll always use <!DOCTYPE html>.

Open the index.html file created earlier in your text editor and add <!DOCTYPE html> to the very first line.

**HTML ELEMENT**

After we declare the doctype, we need to provide an <html> element. This is what’s known as the root element of the document, meaning that every other element in the document will be a descendant of it.

This becomes more important later on when we learn about manipulating HTML with JavaScript. For now, just know that the <html> element should be included on every HTML document.

Back in the index.html file, let’s add the <html> element by typing out its opening and closing tags, like so:

<!DOCTYPE html>

<html lang="en">

</html>

Noticed the word lang here? It represents an HTML attribute which is associated with the given HTML tag i.e. <html> in this case. These attributes provide additional information about HTML elements. (More about HTML attributes in the following lesson.)

*What is the lang attribute?*

lang specifies the language of the text content in that element. This attribute is primarily used for improving accessibility of the webpage. It allows assistive technologies, for example screen readers, to adapt according to the language and invoke correct pronunciation.

**HEAD ELEMENT**

The <head> element is where we put important meta-information about our webpages, and stuff required for our webpages to render correctly in the browser. Inside the <head>, we should not use any element that displays content on the webpage.

**META ELEMENT**

We should always have the <meta> tag with the charset encoding of the webpage in the <head> element: <meta charset="utf-8">.

Setting the encoding is very important because it ensures that the webpage will display special symbols and characters from different languages correctly in the browser.

**TITLE ELEMENT**

Another element we should always include in the head of an HTML document is the <title> element:

<title>My First Webpage</title>

The <title> element is used to give webpages a human-readable title which is displayed in our webpage’s browser tab. For example, if you look at the current tab’s name of your browser, it will read “HTML Boilerplate | The Odin Project”; this is the <title> of the current .html file.

If we didn’t include a <title> element, the webpage’s title would default to its file name. In our case that would be index.html, which isn’t very meaningful for users; this would make it very difficult to find our webpage if the user has many browser tabs open.

There are many more elements that can go within the head of an HTML document. However, for now it’s only crucial to know about the two elements we have covered here. We will introduce more elements that go into the head throughout the rest of the curriculum.

Back in our index.html file, let’s add a <head> element with a <meta> element and a title within it. The <head> element goes within the <html> element and should always be the first element under the opening <html> tag:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>My First Webpage</title>

</head>

</html>

**BODY ELEMENT**

The final element needed to complete the HTML boilerplate is the <body> element. This is where all the content that will be displayed to users will go - the text, images, lists, links, and so on.

To complete the boilerplate, add a <body> element to the index.html file. The <body> element also goes within the <html> element and is always below the <head> element, like so:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>My First Webpage</title>

</head>

<body>

</body>

</html>

**VIEWING HTML FILES IN THE BROWSER**

The HTML boilerplate in the index.html file is complete at this point, but how do you view it in the browser? There are a couple of different options:

**Note**: In order to avoid branching our lesson’s instructions to accommodate for all of the differences between browsers, we are going to be using Google Chrome as our primary browser for the remainder of this course. All references to the browser will pertain specifically to Google Chrome. We strongly suggest that you use Google Chrome for all of your testing going forward.

1. You can drag and drop an HTML file from your text editor into the address bar of your browser.
2. You can find the HTML file in your file system and then double click it. This will open up the file in the default browser your system uses.
3. You can use the terminal to open the file in your browser.

* Ubuntu - Navigate to the directory containing the file and type google-chrome index.html
* macOS - Navigate to the directory containing the file and type open ./index.html

Using one of the methods above, open up the index.html file we have been working on. You’ll notice the screen is blank. This is because we don’t have anything in our body to display.

Back in the index.html file, let’s add a heading (more on these later) to the body, and save the file:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>My First Webpage</title>

</head>

<body>

<h1>Hello World!</h1>

</body>

</html>

Now, if you refresh the page in the browser, you should see the changes take effect, and the heading “Hello World!” will be displayed.

**VSCODE SHORTCUT**

VSCode has a built-in shortcut you can use for generating all the boilerplate in one go. Please note that this shortcut only works while editing a file with the .html extension or a text file with the HTML language already selected. To trigger the shortcut, delete everything in the index.html file and just enter ! on the first line. This will bring up a couple of options. Press the Enter key to choose the first one, and voila, you should have all the boilerplate populated for you.

You may notice that the boilerplate produced by this shortcut includes a line we have not yet mentioned:

<meta name="viewport" content="width=device-width, initial-scale=1.0">

This is not something we need to know about until we discuss responsive design, an advanced topic involving different screen sizes which we will cover much later in the curriculum. For now, you can leave that line as it is.

It’s still good to know how to write the boilerplate yourself in case you find yourself using a text editor like notepad (heaven forbid), which doesn’t have this shortcut. Try not to use the shortcut in your first few HTML projects, so you can build some muscle memory for writing the boilerplate code.

**TEXT IN HTML**

Most content on the web is text-based, so you will find yourself needing to work with HTML text elements quite a bit.

**PARAGRAPHS**

Here's how to write paragraphs in HTML:

**The p Tag:**

* The fundamental element for paragraphs in HTML is the <p> tag.
* Everything you place between the opening <p> tag and the closing </p> tag will be displayed as a paragraph.

**Basic Structure:**

HTML

<p>This is some text in a paragraph.</p>

This code will display the text "This is some text in a paragraph." as a paragraph on your webpage.

**HEADINGS IN HTML**

Headings are essential for structuring your web page content and improving readability for both users and search engines. Here's a breakdown of headings in HTML:

**HTML Headings: Structure and Semantics**

HTML provides six levels of headings, denoted by the <h1> to <h6> tags. These headings create a hierarchical structure for your web page content, similar to an outline.

* <h1>: The most important heading, typically used for the main page title. There should generally only be one <h1> on a webpage.
* <h2> to <h6>: Subsequent headings for sub-sections and subtopics within the main content. Use these headings in a logical progression to organize your content.

**Here's an example of a basic HTML structure with headings:**

HTML

<!DOCTYPE **html**>

<html>

<head>

<title>My Webpage</title>

</head>

<body>

<h1>Welcome to My Page!</h1>

<h2>Introduction</h2>

<p>This is the introductory paragraph of my webpage.</p>

<h2>Main Content</h2>

<p>Here's the main content of the page, further divided into sections using subheadings:</p>

<h3>Section 1</h3>

<p>Content for section 1.</p>

<h3>Section 2</h3>

<p>Content for section 2.</p>

</body>

</html>

**STRONG ELEMENT**

The <strong> element makes text bold. It also semantically marks text as important; this affects tools, like screen readers, that users with visual impairments will rely on to use your website. The tone of voice on some screen readers will change to communicate the importance of the text within a strong element. To define a strong element we wrap text content in a <strong> tag.

**EM ELEMENT**

The <em> element makes text italic. It also semantically places emphasis on the text, which again may affect things like screen readers. To define an emphasised element we wrap text content in an <em> tag.

**NESTING AND INDENTATION**

Nesting and indentation are fundamental concepts in HTML that go hand-in-hand. They work together to create a well-structured and readable web page.

**Nesting: Organizing Your HTML Elements**

Imagine your web page as a collection of building blocks. These building blocks are HTML elements, such as headings (<h1>), paragraphs (<p>), images (<img>), and more. Nesting allows you to create a hierarchy within your code by placing one element inside another.

Here's an example:

HTML

<h1>My Website</h1>

<p>This is the main content of my website.</p>

<ul>

<li>Item 1</li>

<li>Item 2</li>

<li>

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

</li>

</ul>

* The <h1> element acts as the main heading.
* The <p> element containing the main content is nested **within** the <h1> element.
* An unordered list (<ul>) is nested after the paragraph.
* Each list item (<li>) is nested within the <ul> element.
* Notice how a link (<a>) is further nested **inside** one of the list items.

**Indentation: Enhancing Readability**

While nesting defines the hierarchical structure, indentation helps visualize that structure in your code. Proper indentation makes your HTML code easier to read and understand. Here's how:

* **Consistent Indentation:** Use consistent spacing (usually two or four spaces) to indent elements nested within others.
* **Visual Hierarchy:** The indentation reflects the nesting hierarchy. More indented elements are nested deeper within the structure.

Here's the previous example with indentation:

HTML

<h1>My Website</h1>

<p>This is the main content of my website.</p>

<ul>

<li>Item 1</li>

<li>Item 2</li>

<li>

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

</li>

</ul>

The indentation clearly shows how the list items are nested within the unordered list, and the link is nested within a specific list item.

**Benefits of Nesting and Indentation:**

* **Improved Code Maintainability:** Properly nested and indented code is easier to maintain and update, both for you and other developers.
* **Easier Debugging:** Clear structure simplifies debugging by allowing you to identify where potential errors might reside within the nested elements.
* **Collaboration:** Consistent formatting promotes collaboration as everyone working on the code can understand its structure at a glance.

**COMMENTS**

Comments in HTML are essential for adding explanatory notes to your code, improving readability, and aiding collaboration. While comments are not displayed on the web page itself, they are visible when viewing the HTML source code.

**Why Use Comments?**

* **Clarity and Explanation:** Comments allow you to explain complex code sections, the purpose of specific elements, or reasoning behind certain choices. This makes your code easier to understand for yourself and others who might work on it later.
* **Improved Maintainability:** As websites evolve, comments help developers understand the original intent and functionality of the code. This is especially helpful when revisiting or debugging older code.
* **Collaboration:** Comments can act as instructions or notes for other developers working on the same project, promoting better teamwork and knowledge sharing.

**How to Write Comments in HTML:**

 <! –– and the comment closes with ––>

**Best Practices for Comments:**

* **Clarity and Conciseness:** Strive for clear and concise comments that accurately explain the code's functionality.
* **Avoid Obvious Comments:** Don't comment on things that are self-explanatory in the code itself.
* **Use Descriptive Names:** Meaningful variable and element names can often reduce the need for excessive comments.
* **Document Assumptions:** If your code relies on specific assumptions or external factors, document them in comments to avoid confusion.

**HTML LISTS**

Lists are a fundamental way to organize information in HTML. They allow you to present items in a bulleted or numbered format, improving readability and user experience. Here's a comprehensive look at lists in HTML:

**Types of Lists in HTML:**

HTML offers three main types of lists to suit different presentation needs:

1. **Unordered Lists (Bulleted Lists):** Use the <ul> (unordered list) tag to create a bulleted list. Each item within the list is defined using the <li> (list item) tag. The bullet style (disc, circle, square, etc.) can be customized with CSS, but common browsers typically use black circles by default.

HTML

<ul>

<li>Coffee</li>

<li>Tea</li>

<li>Milk</li>

</ul>

1. **Ordered Lists (Numbered Lists):** Use the <ol> (ordered list) tag to create a numbered list. Similar to unordered lists, each item is defined using the <li> tag. By default, browsers display numbers (1, 2, 3...) in sequence, but you can customize the numbering style (lowercase letters, uppercase letters, Roman numerals) using CSS.

HTML

<ol>

<li>Buy groceries</li>

<li>Cook dinner</li>

<li>Wash dishes</li>

</ol>

**Nesting Lists:**

You can nest lists within other lists to create more complex hierarchies. Simply place the inner list's <ul> or <ol> tags within the <li> tag of the outer list.

**List Attributes:**

* type**attribute (for**<ol>**):** This attribute allows you to specify the numbering style for ordered lists. Common values include "1" (default decimal), "a" (lowercase letters), "A" (uppercase letters), "i" (lowercase Roman numerals), and "I" (uppercase Roman numerals).

**Using Lists Effectively:**

* **Clarity and Structure:** Use lists to present items that are naturally grouped or sequential. This improves readability and helps users scannable your content.
* **Accessibility:** Lists can be helpful for users with screen readers or assistive technologies.
* **Visual Style:** While HTML defines the structure, CSS can be used to style the appearance of lists, such as font size, bullet style, or numbering format.

By effectively using lists in your HTML code, you can create well-organized and user-friendly web pages that present information clearly and efficiently.

**LINKS IN HTML**

Links, also known as hyperlinks, are a cornerstone of the web, allowing users to navigate between web pages and resources. In HTML, links are created using the <a> (anchor) element. Here's a detailed explanation of how to create links in HTML:

**The** <a> **Anchor Tag:**

The <a> tag defines a hyperlink. It has two essential attributes:

1. href **(Hypertext Reference):** This attribute specifies the destination URL of the link. When a user clicks the link, their browser fetches the content from the specified URL.
2. **Link Text:** The text content between the opening and closing <a> tags defines the visible clickable portion of the link. This text should clearly describe the destination or action associated with the link.

**Basic Link Structure:**

HTML

<a href="https://www.example.com">Click Here</a>

In this example, "Click Here" is the clickable text, and "[https://www.example.com](https://www.example.com/)" is the URL the user will be directed to when they click the link.

**Opening Links in New Tabs/Windows:**

By default, clicking a link opens the destination webpage in the same browser tab. You can use the target attribute with the <a> tag to specify how the link should open:

* target="\_blank": Opens the linked page in a new browser tab or window.
* target="\_self": Opens the linked page in the same tab (default behaviour).
* target="\_parent": Opens the linked page in the parent frame (useful for framed webpages).

**Example:**

HTML

<a href="https://www.example.com" target="\_blank">Open in New Tab</a>

**Links to Local Files and Other Resources:**

Links aren't limited to external websites. You can also link to local files (like images or PDFs) within your project directory. Just use the relative path to the file as the href attribute value.

**Example (linking to an image):**

HTML

<a href="images/banner.jpg">View Image</a>

**Links to Email Addresses:**

You can create links that open a user's default email client with a pre-filled recipient address. Use the mailto: scheme in the href attribute:

HTML

<a href="mailto:johndoe@example.com">Send Email</a>

**Best Practices for Links:**

* **Clear and Descriptive Link Text:** Choose link text that accurately reflects the destination or action of the link. This helps users understand where they're going when they click.
* **Accessibility:** Use the title attribute to provide additional descriptive text for screen readers or when hovering over the link.
* **Avoid Broken Links:** Ensure all your links point to valid and existing URLs. Broken links create a poor user experience.

**HTML IMAGES**

Images are essential for adding visual interest, improving engagement, and conveying information on web pages. Here's a breakdown of how to insert images using HTML:

**The** <img> **Tag:**

The <img> tag is the primary way to embed images in HTML documents. It's a self-closing tag, meaning it doesn't require a separate closing tag. However, two attributes are essential for proper functionality:

1. src **(source):** This attribute specifies the path to the image file. The path can be an absolute URL (pointing to an image on a web server) or a relative URL (pointing to an image within your project directory).
2. alt **(alternate text):** This attribute provides alternative text for the image. It's crucial for accessibility, as it describes the image's content for screen readers or if the image fails to load.

**Basic Image Syntax:**

HTML

<img src="images/banner.jpg" alt="Company Logo">

In this example, "images/banner.jpg" is the path to the image file, and "Company Logo" is the alternative text description.

**Additional Attributes:**

* width **and** height**:** These attributes specify the image's width and height in pixels. While not strictly necessary, they can help prevent layout issues while the image loads. Browsers may resize the image to fit the available space if these attributes are omitted.
* title**:** This attribute provides optional tooltip text that appears when a user hovers over the image.

**Best Practices for Images:**

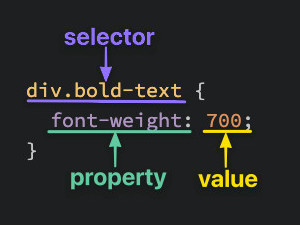
* **Optimize Image Size:** Use image optimization tools to reduce file size without sacrificing quality. This improves page loading speed.
* **Descriptive Alt Text:** Provide meaningful alt text that accurately describes the image's content. This is crucial for accessibility and SEO (Search Engine Optimization).
* **Consider Responsive Design:** Use techniques like responsive images or CSS to ensure your images display well on different screen sizes and devices.

**INTRODUCTION TO CSS**

In the previous lesson, you learned how to write the HTML that determines how a web page is structured. The next step is to make that structure look good with some style, which is exactly what CSS is for. In the next few lessons, we’re going to focus on what we believe are some foundational CSS concepts, things that everyone should know from the beginning — whether they are just starting out or need a refresher.

**Basic syntax**

At the most basic level, CSS is made up of various rules. These rules are made up of a selector (more on this in a bit) and a semicolon-separated list of declarations, with each of those declarations being made up of a property–value pair.

****

**Note**

A <div> is one of the basic HTML elements. It is an empty container. In general, it is best to use other tags such as <h1> or <p> for content in your projects, but as we learn more about CSS you’ll find that there are many cases where the thing you need is just a container for other elements. Many of our exercises use plain<div>s for simplicity. Later lessons will go into much more depth about when it is appropriate to use the various HTML elements.

**Selectors**

Selectors refer to the HTML elements to which CSS rules apply; they’re what is actually being “selected” for each rule. The following subsections don’t cover every selector available, but they’re by far the most common and the ones you should get comfortable using first.

**Universal selector**

The universal selector will select elements of any type, hence the name “universal”, and the syntax for it is a simple asterisk. In the example below, every element would have the color: purple; style applied to it.

\* {

color: purple;

}

**Type selectors**

A type selector (or element selector) will select all elements of the given element type, and the syntax is just the name of the element:

<!-- index.html -->

<div>Hello, World!</div>

<div>Hello again!</div>

<p>Hi...</p>

<div>Okay, bye.</div>

/\* styles.css \*/

div {

color: white;

}

Here, all three <div> elements would be selected, while the <p> element wouldn’t be.

Class selectors

Class selectors will select all elements with the given class, which is just an attribute you place on an HTML element. Here’s how you add a class to an HTML tag and select it in CSS:

<!-- index.html -->

<div class="alert-text">Please agree to our terms of service. </div>

/\* styles.css \*/

.alert-text {

color: red;

}

Note the syntax for class selectors: a period immediately followed by the case-sensitive value of the class attribute. Classes aren’t required to be specific to a particular element, so you can use the same class on as many elements as you want.

Another thing you can do with the class attribute is to add multiple classes to a single element as a space-separated list, such as class="alert-text severe-alert". Since whitespace is used to separate class names like this, you should never use spaces for multi-worded names and should use a hyphen instead.

**ID SELECTORS**

ID selectors are similar to class selectors. They select an element with the given ID, which is another attribute you place on an HTML element. The major difference between classes and IDs is that an element can only have one ID. It cannot be repeated on a single page and should not contain any whitespace:

<!-- index.html -->

<div id="title">My Awesome 90's Page</div>

/\* styles.css \*/

#title {

background-color: red;

}

For IDs, instead of a period, we use a hashtag immediately followed by the case-sensitive value of the ID attribute. A common pitfall is people overusing the ID attribute when they don’t necessarily need to, and when classes will suffice. While there are cases where using an ID makes sense or is needed, such as taking advantage of specificity or having links redirect to a section on the current page, you should use IDs sparingly (if at all).

**THE GROUPING SELECTOR**

What if we have two groups of elements that share some of their style declarations?

.read {

color: white;

background-color: black;

/\* several unique declarations \*/

}

.unread {

color: white;

background-color: black;

/\* several unique declarations \*/

}

Both our .read and .unread selectors share the color: white; and background-color: black; declarations, but otherwise have several of their own unique declarations. To cut down on the repetition, we can group these two selectors together as a comma-separated list:

.read,

.unread {

color: white;

background-color: black;

}

.read {

/\* several unique declarations \*/

}

.unread {

/\* several unique declarations \*/

}

Both of the examples above (with and without grouping) will have the same result, but the second example reduces the repetition of declarations and makes it easier to edit either the color or background-color for both classes at once.

**CHAINING SELECTORS**

Another way to use selectors is to chain them as a list without any separation. Let’s say we had the following HTML:

<div>

<div class="subsection header">Latest Posts</div>

<p class="subsection preview">This is where a preview for a post might go.</p>

</div>

We have two elements with the subsection class that have some sort of unique styles, but what if we only want to apply a separate rule to the element that also has header as a second class? Well, we could chain both the class selectors together in our CSS like so:

.subsection.header {

color: red;

}

What .subsection.header does is it selects any element that has both the subsection and header classes. Notice how there isn’t any space between the .subsection and .header class selectors. This syntax basically works for chaining any combination of selectors, except for chaining more than one type selector.

This can also be used to chain a class and an ID, as shown below:

<div>

<div class="subsection header">Latest Posts</div>

<p class="subsection" id="preview">

This is where a preview for a post might go.

</p>

</div>

You can take the two elements above and combine them with the following:

.subsection.header {

color: red;

}

.subsection#preview {

color: blue;

}

In general, you can’t chain more than one type selector since an element can’t be two different types at once. For example, chaining two type selectors like div and p would give us the selector divp, which wouldn’t work since the selector would try to find a literal <divp> element, which doesn’t exist.

**DESCENDANT COMBINATOR**

Combinators allow us to combine multiple selectors differently than either grouping or chaining them, as they show a relationship between the selectors. There are four types of combinators in total, but for right now we’re going to only show you the descendant combinator, which is represented in CSS by a single space between selectors. A descendant combinator will only cause elements that match the last selector to be selected if they also have an ancestor (parent, grandparent, etc.) that matches the previous selector.

So something like .ancestor .child would select an element with the class child if it has an ancestor with the class ancestor. Another way to think of it is that child will only be selected if it is nested inside ancestor, regardless of how deep that nesting is. Take a quick look at the example below and see if you can tell which elements would be selected based on the CSS rule provided:

<!-- index.html -->

<div class="ancestor">

<!-- A -->

<div class="contents">

<!-- B -->

<div class="contents"><!-- C --></div>

</div>

</div>

<div class="contents"></div>

<!-- D -->

/\* styles.css \*/

.ancestor .contents {

/\* some declarations \*/

}

In the above example, the first two elements with the contents class (B and C) would be selected, but that last element (D) wouldn’t be. Was your guess correct?

There’s really no limit to how many combinators you can add to a rule, so .one .two .three .four would be totally valid. This would just select an element that has a class of four if it has an ancestor with a class of three, and if that ancestor has its own ancestor with a class of two, and so on. You generally want to avoid trying to select elements that need this level of nesting, though, as it can get pretty confusing and long, and it can cause issues when it comes to specificity.

**PROPERTIES TO GET STARTED WITH**

There are some CSS properties that you’re going to be using all the time, or at the very least more often than not. We’re going to introduce you to several of these properties, though this is by no means a complete list. Learning the following properties will be enough to help get you started.

**Color and background-color**

The color property sets an element’s text color, while background-color sets, well, the background color of an element. I guess we’re done here?

Almost. Both of these properties can accept one of several kinds of values. A common one is a keyword, such as an actual color name like red or the transparent keyword. They also accept HEX, RGB, and HSL values, which you may be familiar with if you’ve ever used a photoshop program or a site where you could customize your profile colors.

p {

/\* hex example: \*/

color: #1100ff;

}

p {

/\* rgb example: \*/

color: rgb(100, 0, 127);

}

p {

/\* hsl example: \*/

color: hsl(15, 82%, 56%);

}

Take a quick look at CSS Legal Color Values to see how you can adjust the opacity of these colors by adding an alpha value.

**Typography basics and text-align**

font-family can be a single value or a comma-separated list of values that determine what font an element uses. Each font will fall into one of two categories, either a “font family name” like "Times New Roman" (we use quotes due to the whitespace between words) or a “generic family name” like serif (generic family names never use quotes).

If a browser cannot find or does not support the first font in a list, it will use the next one, then the next one and so on until it finds a supported and valid font. This is why it’s best practice to include a list of values for this property, starting with the font you want to be used most and ending with a generic font family as a fallback, e.g. font-family: "Times New Roman", serif;

font-size will, as the property name suggests, set the size of the font. When giving a value to this property, the value should not contain any whitespace, e.g. font-size: 22px has no space between “22” and “px”.

font-weight affects the boldness of text, assuming the font supports the specified weight. This value can be a keyword, e.g. font-weight: bold, or a number between 1 and 1000, e.g. font-weight: 700 (the equivalent of bold). Usually, the numeric values will be in increments of 100 up to 900, though this will depend on the font.

text-align will align text horizontally within an element, and you can use the common keywords you may have come across in word processors as the value for this property, e.g. text-align: center.

**Image height and width**

Images aren’t the only elements that we can adjust the height and width on, but we want to focus on them specifically in this case.

By default, an <img> element’s height and width values will be the same as the actual image file’s height and width. If you wanted to adjust the size of the image without causing it to lose its proportions, you would use a value of “auto” for the height property and adjust the width value:

img {

height: auto;

width: 500px;

}

For example, if an image’s original size was 500px height and 1000px width, using the above CSS would result in a height of 250px.

These properties work in conjunction with the height and width attributes in the HTML. It’s best to include both of these properties and the HTML attributes for image elements, even if you don’t plan on adjusting the values from the image file’s original ones. When these values aren’t included, if an image takes longer to load than the rest of the page contents, it won’t take up any space on the page at first but will suddenly cause a drastic shift of the other page contents once it does load in. Explicitly stating a height and width prevents this from happening, as space will be “reserved” on the page and appear blank until the image loads.

**ADDING CSS TO HTML**

Now that we’ve learned some basic syntax, you might be wondering how to add all this CSS to our HTML. There are three methods to do so.

External CSS

External CSS is the most common method you will come across, and it involves creating a separate file for the CSS and linking it inside of an HTML’s opening and closing <head> tags with a self-closing <link> element:

<!-- index.html -->

<head>

<link rel="stylesheet" href="styles.css">

</head>

/\* styles.css \*/

div {

color: white;

background-color: black;

}

p {

color: red;

}

First, we add a self-closing <link> element inside of the opening and closing <head> tags of the HTML file. The href attribute is the location of the CSS file, either an absolute URL or, what you’ll be utilizing, a URL relative to the location of the HTML file. In our example above, we are assuming both files are located in the same directory. The rel attribute is required, and it specifies the relationship between the HTML file and the linked file.

Then inside of the newly created styles.css file, we have the selector (the div and p), followed by a pair of opening and closing curly braces, which create a “declaration block”. Finally, we place any declarations inside of the declaration block. color: white; is one declaration, with color being the property and white being the value, and background-color: black; is another declaration.

A note on file names: styles.css is just what we went with as the file name here. You can name the file whatever you want as long as the file type is .css, though “style” or “styles” is most commonly used.

A couple of the pros to this method are:

* It keeps our HTML and CSS separated, which results in the HTML file being smaller and making things look cleaner.
* We only need to edit the CSS in one place, which is especially handy for websites with many pages that all share similar styles.

**INTERNAL CSS**

Internal CSS (or embedded CSS) involves adding the CSS within the HTML file itself instead of creating a completely separate file. With the internal method, you place all the rules inside of a pair of opening and closing <style> tags, which are then placed inside of the opening and closing <head> tags of your HTML file. Since the styles are being placed directly inside of the <head> tags, we no longer need a <link> element that the external method requires.

Besides these differences, the syntax is exactly the same as the external method (selector, curly braces, declarations):

<head>

<style>

div {

color: white;

background-color: black;

}

p {

color: red;

}

</style>

</head>

<body>

...

</body>

This method can be useful for adding unique styles to a single page of a website, but it doesn’t keep things separate like the external method, and depending on how many rules and declarations there are it can cause the HTML file to get pretty big.

Inline CSS

Inline CSS makes it possible to add styles directly to HTML elements, though this method isn’t as recommended:

<body>

<div style="color: white; background-color: black;">...</div>

</body>

The first thing to note is that we don’t actually use any selectors here, since the styles are being added directly to the opening <div> tag itself. Next, we have the style= attribute, with its value within the pair of quotation marks being the declarations.

If you need to add a unique style for a single element, this method can work just fine. Generally, though, this isn’t exactly a recommended way for adding CSS to HTML for a few reasons:

* It can quickly become pretty messy once you start adding a lot of declarations to a single element, causing your HTML file to become unnecessarily bloated.
* If you want many elements to have the same style, you would have to copy and paste the same style to each individual element, causing lots of unnecessary repetition and more bloat.
* Any inline CSS will override the other two methods, which can cause unexpected results. (While we won’t dive into it here, this can actually be taken advantage of.)

**INSPECTING HTML AND CSS**

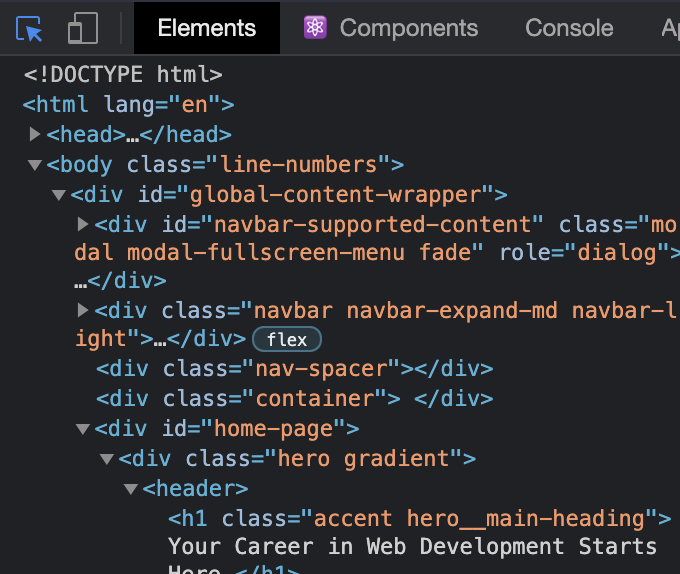
**The inspector**

To open up the inspector, you can right-click on any element of a webpage and click “Inspect” or press F12. Go ahead and do that right now to see the HTML and CSS used on this page.

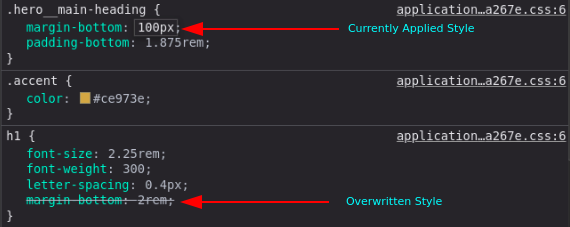
Don’t get overwhelmed with all the tools you’re now seeing! For this lesson, we want to focus on the Elements and Styles panels.

**Inspecting elements**

In the Elements panel, you can see the entire HTML structure of your page. You can click on any of the elements in this panel to select that specific element. Alternatively, you can click the blue-highlighted icon shown below on the left, and hover over any element on the page.

[](https://cdn.statically.io/gh/TheOdinProject/curriculum/594984d7c9f9e744577f19ea475b3864e8cc7c91/html_css/v2/foundations/inspecting-html-and-css/imgs/01.png)

When an element is selected, the Styles tab will show all the currently applied styles, as well as any styles that are being overwritten (indicated by a strikethrough of the text). For example, if you use the inspector to click on the “Your Career in Web Development Starts Here” header on the TOP homepage, on the right-hand side you’ll see all the styles that are currently affecting the element, as seen below:

[](https://cdn.statically.io/gh/TheOdinProject/curriculum/f8fd38fc62578d8e8368f5303126215a492847f0/foundations/html_css/inspecting-html-and-css/imgs/03.png)

**Testing styles in the inspector**

The Styles panel also allows you to edit styles directly in the browser. You can click inside of any individual selector to add a new rule or click on an existing attribute or value to alter it. When doing so, the webpage responds with the changes in real-time. This won’t affect the source code in your text editor, but it is extremely useful for quickly testing out various attributes and values without needing to reload the page over and over again.

**The Box Model**

**Introduction**

Now that you understand the basic syntax of HTML and CSS, we’re going to get serious. The most important skills you need to master with CSS are positioning and layout. Changing fonts and colors is a crucial skill, but being able to put things exactly where you want them on a webpage is even more crucial. After all, how many webpages can you find where absolutely every element is just stacked one on top of another?

Learning to position elements on a webpage is not that difficult once you understand just a few key concepts. Unfortunately, many learners race through learning HTML and CSS to get to JavaScript and end up missing these fundamental concepts. This leads to frustration, pain, and confusion because all the JavaScript skills in the world are meaningless if you can’t stick your elements on the page where you need them to be. So with that in mind, let’s get started.

**The box model**

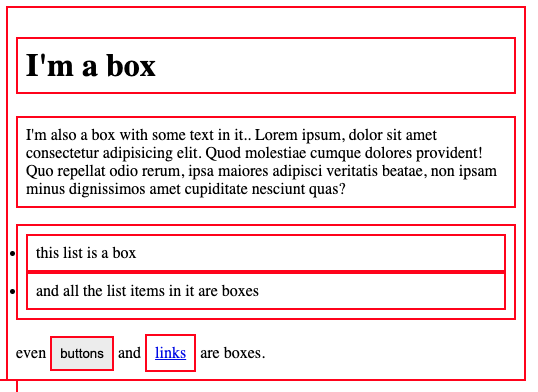
The first important concept that you need to understand to be successful in CSS is the box model. It isn’t complicated, but skipping over it now would cause you much frustration down the line.

Every single thing on a webpage is a rectangular box. These boxes can have other boxes in them and can sit alongside one another. You can get a rough idea of how this works by applying an outline to every element on the page like this:

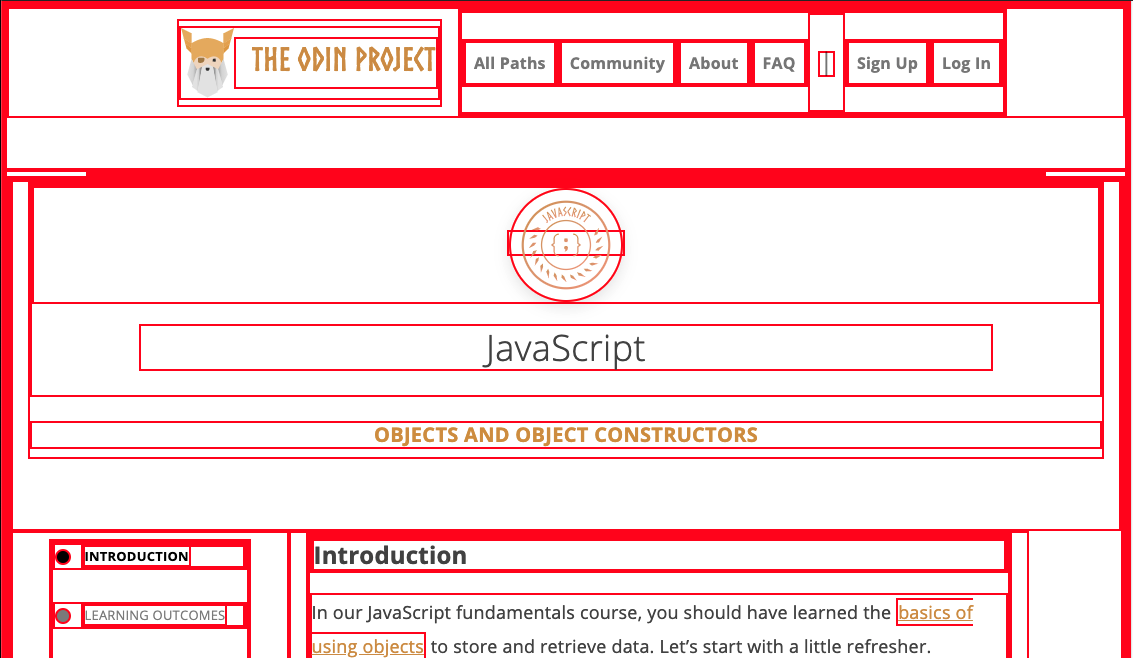
\* {

outline: 2px solid red;

}

[](https://cdn.statically.io/gh/TheOdinProject/curriculum/main/foundations/html_css/css-foundations/the-box-model/imgs/boxes.png)

You can use the browser’s inspector to add the CSS above to this web page if you want. Boxes in boxes!

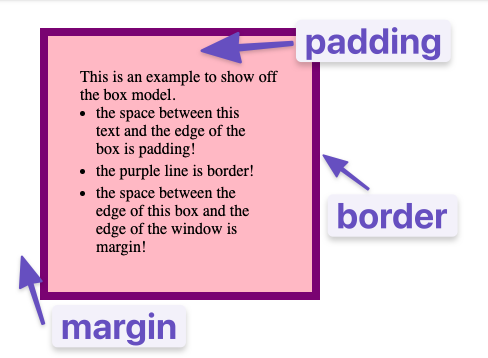
[](https://cdn.statically.io/gh/TheOdinProject/curriculum/main/foundations/html_css/css-foundations/the-box-model/imgs/odin-lined.png)

OK, so there might be some circles in the above image… but when it comes to layout, they fit together like rectangular boxes and not circles. In the end, laying out a webpage and positioning all its elements is deciding how you are going to nest and stack these boxes.

The only real complication here is that there are many ways to manipulate the size of these boxes, and the space between them, using padding, margin, and border. The assigned articles go into more depth on this concept, but to sum it up briefly:

* padding increases the space between the border of a box and the content of the box.
* margin increases the space between the borders of a box and the borders of adjacent boxes.
* border adds space (even if it’s only a pixel or two) between the margin and the padding.

Be sure to study the diagrams carefully.

[](https://cdn.statically.io/gh/TheOdinProject/curriculum/main/foundations/html_css/css-foundations/the-box-model/imgs/box-model.png)

**BLOCK AND INLINE**

**Introduction**

In the previous lesson, we discovered that different display types have unique box models, and we can modify the box calculation by changing the display property. CSS has two box types: block and inline boxes, which determine element behavior and interaction. The display property controls how HTML elements appear on the webpage. We will explore its various options further in this lesson.

**Lesson overview**

This section contains a general overview of topics that you will learn in this lesson.

* You’ll learn about “Normal flow”.
* You’ll learn the difference between block and inline elements.
* You’ll learn which elements default to block and which elements default to inline.
* You’ll learn what divs and spans are.

**Block vs inline**

Most of the elements that you have learned about so far are block elements. In other words, their default style is display: block. By default, block elements will appear on the page stacked atop each other, each new element starting on a new line.

Inline elements, however, do not start on a new line. They appear in line with whatever elements they are placed beside. A clear example of an inline element is a link, or <a> tag. If you stick one of these in the middle of a paragraph of text, the link will behave like a part of the paragraph. Additionally, padding and margin behave differently on inline elements. In general, you do not want to try to put extra padding or margin on inline elements.

The middle ground inline-block

Inline-block elements behave like inline elements, but with block-style padding and margin. display: inline-block is a useful tool to know about, but in practice, you’ll probably end up reaching for flexbox more often if you’re trying to line up a bunch of boxes. Flexbox will be covered in-depth in the next lesson.

**Divs and spans**

We can’t talk about block and inline elements without discussing divs and spans. All the other HTML elements we have encountered so far give meaning to their content. For example, paragraph elements tell the browser to display the text it contains as a paragraph. Strong elements tell the browser which texts within are important and so on. Yet, divs and spans give no particular meaning to their content. They are just generic boxes that can contain anything.

Having elements like this available to us is a lot more useful than it may first appear. We will often need elements that serve no other purpose than to be “hook” elements. We can give an id or class to target them for styling with CSS. Another use case we will face regularly is grouping related elements under one parent element to correctly position them on the page. Divs and spans provide us with the ability to do this.

Div is a block-level element by default. It is commonly used as a container element to group other elements. Divs allow us to divide the page into different blocks and apply styling to those blocks.

**FLEXBOX BASICS**

Flexbox is a one-dimensional layout model for arranging elements along a container.

It offers more control over element placement compared to traditional methods (floats, inline-block).

It uses a container (flex container) and its child elements (flex items).

**Terminology:**

Main Axis: The primary axis along which flex items are arranged (horizontal by default).

Cross Axis: The axis perpendicular to the main axis (vertical by default).

Flex Container: The parent element that holds the flex items.

Flex Items: The child elements within the flex container that are laid out using flexbox.

**Creating a Flex Container:**

Set the display property of the container element to flex.

CSS

.flex-container {

display: flex;

}

**Aligning Flex Items:**

justify-content: This property controls the alignment of flex items along the main axis. Here are some common options:

flex-start (default): Align items to the beginning of the container.

flex-end: Align items to the end of the container.

center: Center items within the container.

space-between: Distribute items evenly with space between them.

space-around: Distribute items with equal space around them (including before and after).

**Apply justify-content to the flex container:**

CSS

.flex-container {

display: flex;

justify-content: space-between;

}

**Aligning Items on the Cross Axis:**

align-items: This property controls the alignment of flex items along the cross axis. Here are some common options:

flex-start (default): Align items to the top of the container.

flex-end: Align items to the bottom of the container.

center: Center items on the cross axis.

baseline: Align items based on their baselines (useful for text).

**Apply align-items to the flex container:**

.flex-container {

display: flex;

justify-content: space-between;

align-items: center;

}

**Flex Item Sizing:**

By default, flex items will share the available space within the container.

You can control the size of flex items using:

flex-grow: Controls how much extra space an item can grow compared to others (default: 0).

flex-shrink: Controls how much an item can shrink to accommodate other items (default: 1).

flex-basis: Sets the default size of an item (can be a percentage or pixel value).

Example:

CSS

.flex-container {

display: flex;

justify-content: space-between;

align-items: center;

}

.item1 {

flex: 2; /\* Grows twice as much as item2 \*/

background-color: #f0f0f0;

padding: 10px;

}

.item2 {

flex: 1; /\* Grows half as much as item1 \*/

background-color: #ddd;

padding: 10px;

}

**Growing and Shrinking**

Let’s look a little closer at what actually happened when you put flex: 1 on those flex items in the last lesson.

**LESSON OVERVIEW**

This section contains a general overview of topics that you will learn in this lesson.

You’ll learn the 3 properties that are defined by the flex shorthand, and how to use them individually.

**THE FLEX SHORTHAND**

The flex declaration is actually a shorthand for 3 properties that you can set on a flex item. These properties affect how flex items size themselves within their container. You’ve seen some shorthand properties before, but we haven’t officially defined them yet.

Shorthand properties are CSS properties that let you set the values of multiple other CSS properties simultaneously. Using a shorthand property, you can write more concise (and often more readable) stylesheets, saving time and energy.

Source: Shorthand properties on MDN

In this case, flex is actually a shorthand for flex-grow, flex-shrink and flex-basis.

In the above screenshot, flex: 1 equates to: flex-grow: 1, flex-shrink: 1, flex-basis: 0.

Very often you see the flex shorthand defined with only one value. In that case, that value is applied to flex-grow. So when we put flex: 1 on our divs, we were actually specifying a shorthand of flex: 1 1 0.

**Flex-grow**

flex-grow expects a single number as its value, and that number is used as the flex-item’s “growth factor”. When we applied flex: 1 to every div inside our container, we were telling every div to grow the same amount. The result of this is that every div ends up the exact same size. If we instead add flex: 2 to just one of the divs, then that div would grow to 2x the size of the others.

In the following example the flex shorthand has values for flex-shrink and flex-basis specified with their default values.

**Flex-shrink**

flex-shrink is similar to flex-grow, but sets the “shrink factor” of a flex item. flex-shrink only ends up being applied if the size of all flex items is larger than their parent container. For example, if our 3 divs from above had a width declaration like: width: 100px, and .flex-container was smaller than 300px, our divs would have to shrink to fit.

The default shrink factor is flex-shrink: 1, which means all items will shrink evenly. If you do not want an item to shrink then you can specify flex-shrink: 0;. You can also specify higher numbers to make certain items shrink at a higher rate than normal.

Here’s an example. Note that we’ve also changed the flex-basis for reasons that will be explained shortly. If you shrink your browser window you’ll notice that .two never gets smaller than the given width of 250px, even though the flex-grow rule would otherwise specify that each element should be equally sized.

An important implication to notice here is that when you specify flex-grow or flex-shrink, flex items do not necessarily respect your given values for width. In the above example, all 3 divs are given a width of 250px, but when their parent is big enough, they grow to fill it. Likewise, when the parent is too small, the default behavior is for them to shrink to fit. This is not a bug, but it could be confusing behavior if you aren’t expecting it.

**FLEX-BASIS**

flex-basis sets the initial size of a flex item, so any sort of flex-growing or flex-shrinking starts from that baseline size. The shorthand value defaults to flex-basis: 0%. The reason we had to change it to auto in the flex-shrink example is that with the basis set to 0, those items would ignore the item’s width, and everything would shrink evenly. Using auto as a flex-basis tells the item to check for a width declaration (width: 250px).

**Important note about flex-basis:**

There is a difference between the default value of flex-basis and the way the flex shorthand defines it if no flex-basis is given. The actual default value for flex-basis is auto, but when you specify flex: 1 on an element, it interprets that as flex: 1 1 0. If you want to only adjust an item’s flex-grow you can do so directly, without the shorthand. Or you can be more verbose and use the full 3 value shorthand flex: 1 1 auto, which is also equivalent to using flex: auto.

**WHAT IS FLEX: AUTO?**

If you noticed, we mentioned a new flex shorthand flex: auto in the previous note. However we didn’t fully introduce it. flex: auto is one of the shorthands of flex. When auto is defined as a flex keyword it is equivalent to the values of flex-grow: 1, flex-shrink: 1 and flex-basis: auto or to flex: 1 1 auto using the flex shorthand. Note that flex: auto is not the default value when using the flex shorthand despite the name being “auto” which may be slightly confusing at first. You will encounter and learn more about flex: auto and its potential use-cases when reading through the assignment section.

In practice…

In practice you will likely not be using complex values for flex-grow, flex-shrink or flex-basis. Generally, you’re most likely to use declarations like flex: 1; to make divs grow evenly and flex-shrink: 0 to keep certain divs from shrinking.

It is possible to get fancy, and set up layouts where some columns relate to each other in a specific ratio, so it’s useful to know that you can use other values, but those are relatively rare.