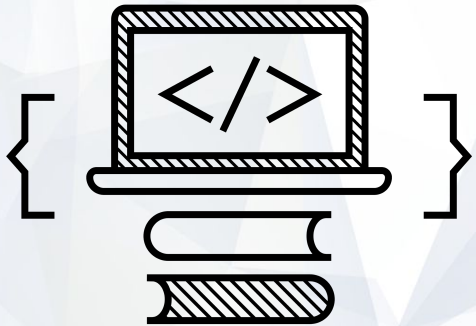
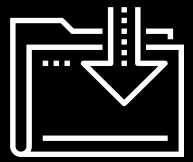


Web, HTML, CSS

Course: Java

S1



Learning Outcomes

By the end of this lesson, you will be able to:

Explain how the web works.

Explain the difference between a website and a web app.

Use the browser's dev tools to edit websites.

Use HTML and CSS to create a simple three-page website.

The Web

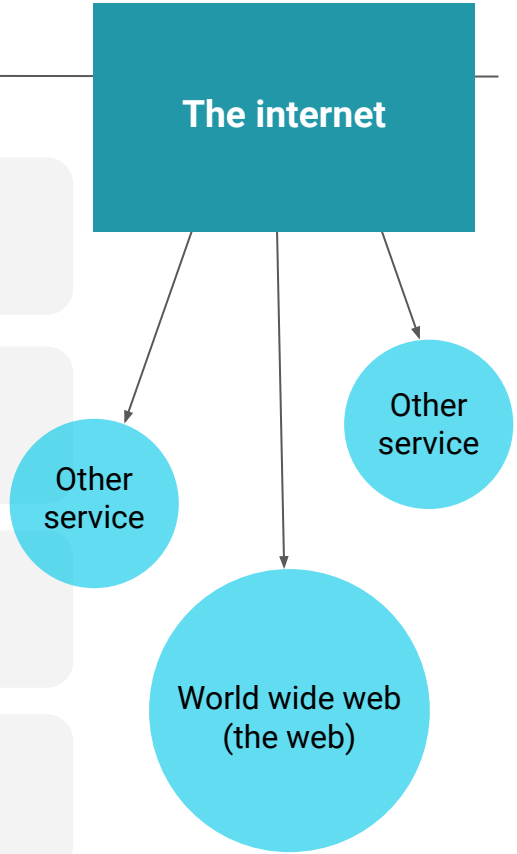
How Does the Web Work?

The web is a subset of the internet. The internet can exist without the web, but not vice versa.

The internet is any node-to-node connection—or, communication between two devices that are capable of encoding and transmitting streams/packets of binary data.

For communication to occur, there must be an agreed-upon standard set of rules, or a **protocol**. Generally, **Transmission Control Protocol (TCP)** is used, in conjunction with **Internet Protocol (IP)**.

The internet has been around in various forms since the 60s. The web, using **HyperText Transfer Protocol (HTTP)**, was created in 1990 by Sir Tim Berners-Lee.



How Does the Web Work? (continued)

HyperText is a reference to text that links/jumps to other text or other media. The web is essentially a group of documents that link to one another.

Web browsers make the web accessible by facilitating HTTP **requests** and **responses**.

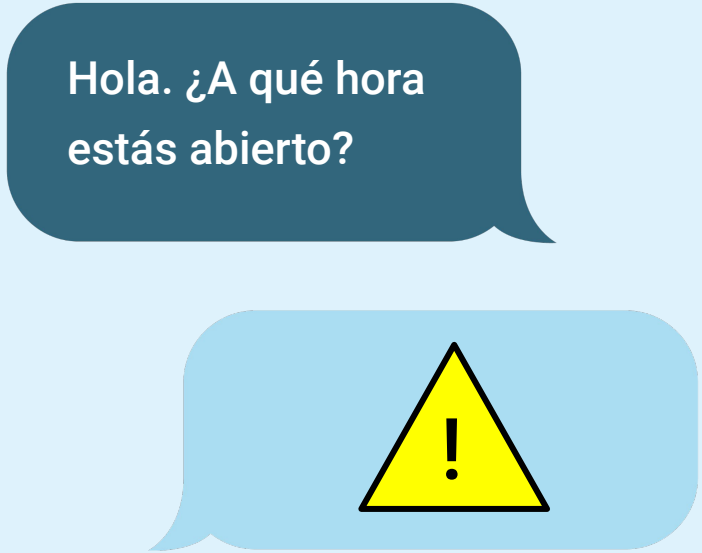
Among other things, responses contain **status codes** (e.g., '200' is 'OK', '404' is 'Not Found') along with a **payload**. Browsers consume the payload to render a page, using a combination of **HyperText Markup Language (HTML)**, **Cascading Style Sheets (CSS)**, and **JavaScript (JS)**.

Phone Analogy

- When we want to call somebody, rather than typing in the phone number directly, we can look them up by name in our Contacts.
- Similarly, when we want to call a website, rather than using the server's IP address directly, we look it up using a Domain Name Server (DNS). This is handled automatically via our Internet Service Provider (ISP).
- On the phone, once we dial the number, we wait for a response. Usually, the person will say, "Hello," or we're greeted with voicemail content. Alternatively, if the number is disconnected, or if there is some other issue, we may hear an error message.

Phone Analogy (continued)

- For a website, we will eventually be “greeted” with some content to look at (or an error message). This encapsulates the **request-response cycle**.
- During a phone call, we can ask questions in order to receive information. But, the person whom we’re talking to might not have the answers to all of our questions, and/or we might ask about something that they don't want to talk about. In order for the conversation to be fruitful, we have to speak the same language and follow some type of protocol.
- Similarly, as we continue to interact with a website, we will continue requesting information and we will keep getting responses, either affirmative or error messages. We will carry on this “conversation” via a series of HyperText Transfer Protocol requests/responses.



Website vs. Web App

Time to Code

Website vs. Web App

Suggested Time:

5 minutes

Dev Tools

Time to Code

Use Dev Tools Inspector

Suggested Time:

10 minutes

Activity: Find Another Website and Use Dev Tools to Change It Up

Suggested Time:

5 minutes

Mobile-First Responsive Web Design

What is Mobile-First Responsive Web Design (RWD)?

- For many users, the mobile display is their first impression of a website.
- So, **mobile-first responsive web design** can be an effective strategy when designing sites for the web.
- But, it's also okay to focus on the desktop view. In fact, many admin panels are meant only for use on larger desktop screens.

Activity: Use the Device Emulators to Test a Website

Suggested Time:

5 minutes


RECAP

Recap

If we open up a browser and type in “cnn.com”, assuming that we have a stable internet connection, what happens next? Describe the process by which our browsers render the HTML, CSS, and JS.

How do we open up the browser's dev tools?

What are some of the things that we can accomplish from the dev tools? Why are they so valuable for web developers?

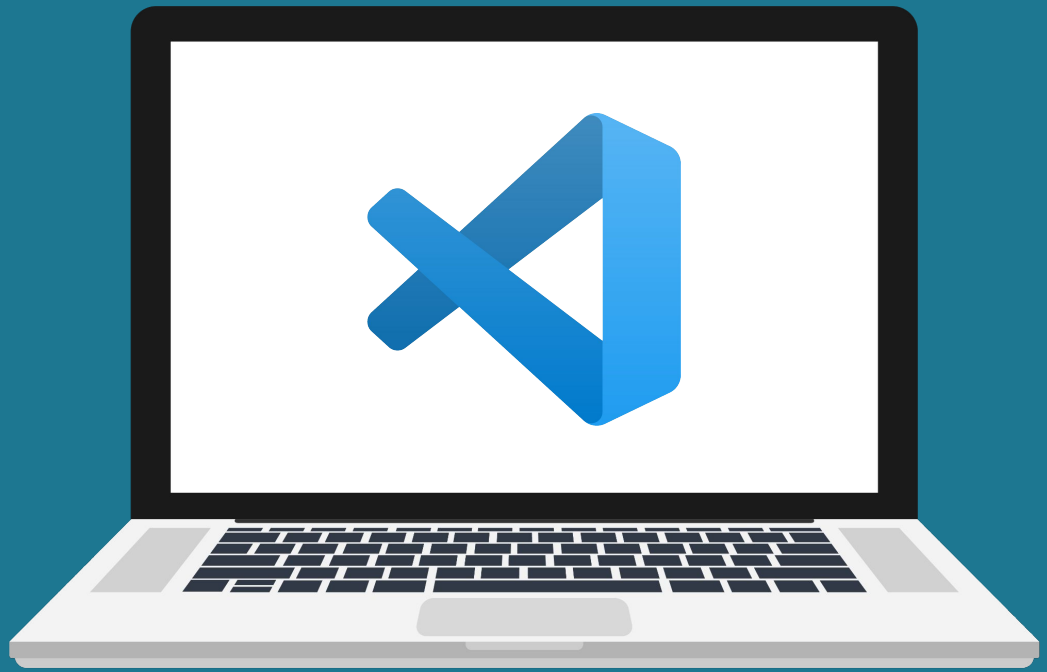
What are the specific roles of HTML, CSS, and JS, respectively?


Break

Dev Tools

Web Dev Tool Kit

- VS Code is a simple yet highly customizable **text editor**.
- It's not quite an IDE, but with the right configuration, it can come close to one.
- VS Code is the de facto open-source standard in the front-end dev community.



Time to Code

Check Out VS Code Site

Suggested Time:

5 mins

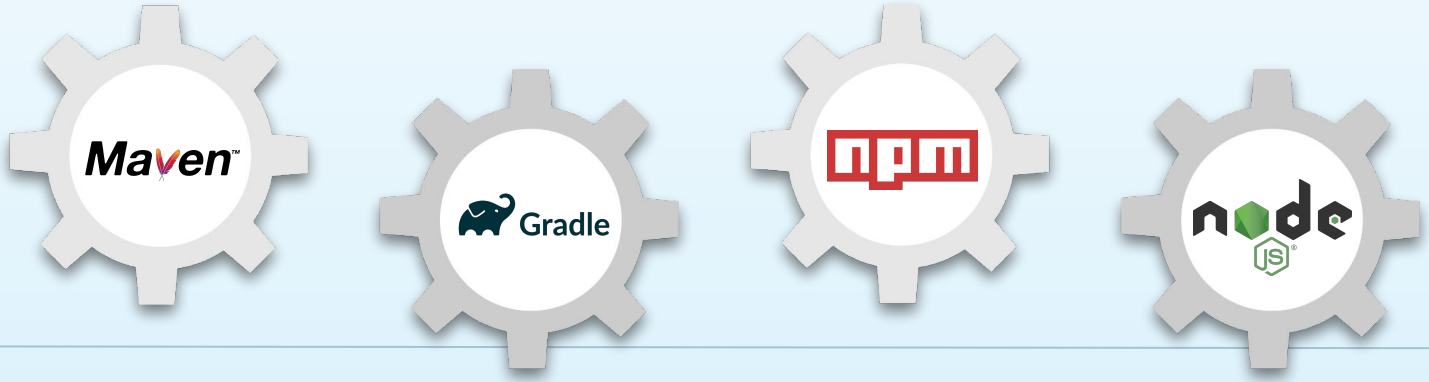
Activity: Install VS Code

Suggested Time:

5 minutes

NodeJS / npm

- In order to be a productive web developer, it's useful to have a reliable set of tools.
- In Java, where we have a “tool shop” such as Maven or Gradle, there is npmjs.com. We will use `npm install` from our terminal to install a suite of code-quality tools that are used widely in the front-end developer community.
- In order to access `npm`, we will install NodeJS.
- Traditionally, JS has only worked within JS engines in various browsers. Around 2011, Ryan Dahl leveraged Chrome's V8 engine to create an environment that allowed developers to write JS code on the server side (outside of the browser).
- NodeJS serves a similar role to Java Swing. Among other things, it will handle incoming HTTP requests and send back HTTP responses.



Time to Code

Explore Common Front-End Web Dev Tooling

Suggested Time:

5 minutes

Activity: Install NodeJS

Suggested Time:

5 minutes

HTML

It defines the meaning and structure of web content.

MDN Web Docs

HTML is for describing content. Generally, we do not use it for any type of element styling.

The Role of HTML and Common Tags

One exception to the previous rule: when we specify `width` on `<img>`. In this case, it allows the browser to reserve space for an image by knowing its intended width up front, resulting in a more efficient render.

HTML code is made up of opening and closing tags that establish a parent-child, or tree-like, structure.

```
<!DOCTYPE html>
<!-- Notice the 'parent-child' relationship or 'tree-like' structure.
Notice the opening and closing tags. -->
<html>
  <head>
    <!-- Metadata, title, etc. goes here -->
  </head>
  <body>
    <!-- Visible content that we want to show the user goes here! -->
  </body>
</html>
```

<head>

The HTML `<head>` element contains machine-readable information (metadata) about the document, such as its title, scripts, and style sheets.

The `<head>` element is generally boilerplate. It keeps meta information regarding our page that can help with search engine optimization (SEO). It keeps the `<title>` and maybe a favicon that appears in the browser tab. It also contains `<link>`s to our CSS and sometimes includes `<script>`s too, although for performance reasons, those are generally kept toward the bottom of our page. (This prevents JS from blocking the page and lets HTML and CSS be painted.)

Nothing that we put in `<head>` will display in the browser window.

<body>

We find `<body>` below `<head>`. It is where we put the content for our page.

This is what a user sees in their browser window.

<!DOCTYPE html>

This, too, is boilerplate, but it signifies that we are writing modern HTML and that the browser should do its best to render the page as such.

<html>

The `<html>` tag wraps both `<head>` and `<body>`.

HTML Comments

We use `<!-- -->` to write **comments** in HTML. In VS Code, you can just hit “CTRL/COMMAND” + “/”.

HTML Attributes

Attributes provide additional specific information to the browser about tags. They are found inside the opening tag, followed by `=` and a value wrapped inside of quotes:

```
<!-- Tell the browser that this page is using the English language -->
<html lang="en">
```

Many attributes are essential, such as `src` inside of `<img>`, while others are there for use with CSS and JS:

```
<!-- An image must specify a source for the image content and should include
some alternate text for accessibility/SEO purposes and/or JIC the image
is broken/missing. -->
<img src="./some-image.jpg" alt="Brief text description of this image" />
```

Accessibility

Ideally, the web ishould be accessible by all, including those with visual impairments. For this reason, we should **always** include the `alt` attribute on any/all `<img>`s.

For advanced accessibility concerns, we may need to apply **Accessible Rich Internet Applications (ARIA)** attributes in some cases, but this is not common. (One of the benefits of using Bootstrap is that these cases are covered for us.)

Self-Closing Tags

Most HTML tags contain some text content (a node—but nothing to do with NodeJS!). In this case, we can expect to find both an opening and a closing tag:

```
<p>I am wrapped up with opening and closing tags!</p>
```

Some content is not meant to wrap any text. We call these tags **self-closing**:

```
<!-- The '/' at the end there is technically optional. -->
<img src="./some-image.jpg" alt="Brief text description of this image"/>
```

Figure

Recall that `<img>` is an inline tag by default. We can wrap an `<img>` inside of a `<figure>` to make it a block. In doing so, we get the option to include a `<figcaption>`—a caption for the figure/image. Sometimes, this `<figcaption>` sufficiently conveys the content of the image, so we don't have to include alt text for the image.

We should always include an `alt` attribute on the `<img>`, but we can leave `alt` as an empty string if the `<figcaption>` sufficiently conveys the image contents:

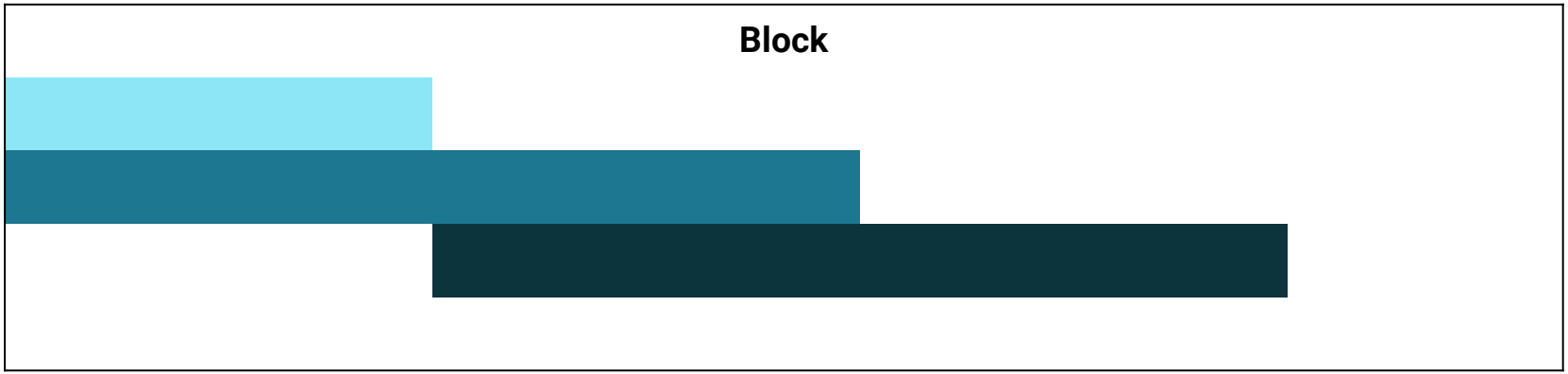
```
<figure>
  <!-- The 'alt' attribute should be present regardless, but we can leave it blank. -->
  <img src="some-image.jpg" alt="" />
  <figcaption>This is some image and we don't need to specify alt if this is
sufficient.</figcaption>
</figure>
```

Block Elements

A **block-level element** occupies the entire horizontal space of its parent element.

It blocks off an entire row and always starts a new line. Block-level elements don't fall inline next to other tags unless they are directed to do so via CSS. If we think of the browser window “row” as a couch...this element stretches out and lays across the whole couch!

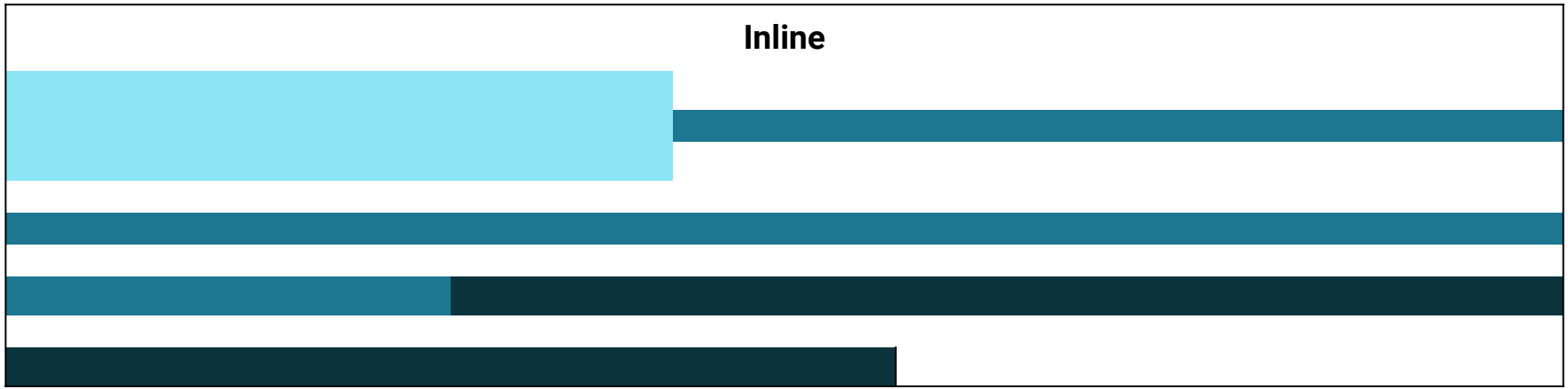
Common examples include `<div>`, `<address>`, `<section>`, `<main>`, `<li>`, `<article>`, `<p>`, etc.



Inline Elements

Unlike block-level elements, **inline elements** “share the couch.” They fall inline, taking up just enough width to show the content that they contain.

Common examples include `<span>`, `<a>`, `<input>`, `<img>`, `<label>`, `<select>`, etc.



The `<br>` tag is not used frequently. It specifies line breaks that are inherently part of the content. Examples of use cases might include a poem or haiku, or a mailing address.

The `<br>` tag should not be used arbitrarily in our HTML to break lines up. This is stylistic and should be a CSS concern.

<hr>

The `<hr>` tag, too, is used infrequently. It represents a thematic break between paragraph-level elements: for example, a change of scene in a story, or a shift of topic within a section.

We should not use these for `border`, which, again, is a stylistic concern better served via CSS.

Email Addresses and Phone Numbers

For email, we prepend `mailto:` in front of the href value:

```
<a href="mailto:hello@somecompany.com">Email Us!</a>
```

For phone, we prepend `tel:` in front of the href value:

```
<a href="tel:+15555555555">(555) 555-5555</a>
```

<address>

The HTML `<address>` element indicates that the enclosed HTML provides contact information for a person, people, or an organization.

It does not have to mean a mailing address, although it might if that is the primary method of communication for users.

```
<address>
  <a href="mailto:hello@somecompany.com">Email Us!</a>
  <a href="tel:+15555555555">(555) 555-5555</a>
</address>
```

Tags for Page Layouts

```
<body>
  <header>
    <!-- TODO: Add an 'h1' and maybe a logo? -->
  </header>

  <main>
    <!-- TODO: Add the bulk of the content that you want your user to see here. -->
  </main>

  <footer>
    <!-- TODO: Add basic copyright information and maybe some social media links here. -->
  </footer>
</body>
```

<header>

- The header content of our page. This might include a logo, `<h1>`, and/or contact information or a call-to-action (CTA) button/link. This is similar to the pre-printed information at the top of a page of company letterhead.

<main>

- The main content of our page. To continue the letterhead analogy, this would be the actual letter content. This tag typically wraps the most amount of content on a page.

Tags for Page Layouts (continued)

<footer>

- The footer content of our page. This might include less important links, social media icons, and/or copyright information. Again, a company letterhead might have something like this pre-printed at the bottom of the page.

** and **

- Both `<div>` and `<span>` are generic tags in that they do not convey any meaning about their contents. They are used as a “hook” for applying CSS in situations where we need to wrap other content blocks, for example.
- Note that `<div>` is `block` and `<span>` is `inline`.
    - The HTML Content Division element (`<div>`) is the generic container for flow content. It has no effect on the content or layout until styled in some way using CSS.
    - The HTML `<span>` element is a generic inline container for phrasing content, which does not inherently represent anything. It can be used to group elements for styling purposes.

<section>

- This is almost as generic as a `<div>`, but it wraps up a section of text.
- It groups blocks of related text. It typically has an `<h2>` inside, followed by `<p>`s.

Tags for Page Layouts (continued)

<article>

- This is a complete document inside of our page. It's usually used for syndicated content that can be shared across multiple sites without losing context/meaning. It's standalone content.

<aside>

- This is tangentially related content. It's optional. An example might be "fun facts" or some other "bonus" content that's not essential reading for the overall message/content of the page.
- Note that this content is not standalone. Unlike `<section>` or `<article>`, it wouldn't make much sense by itself outside of the current webpage.

Tags for Page Layouts (continued)

Heading Tags

- There are six heading levels, `<h1>` through `<h6>`. But, usually, we should probably not go past `<h3>`, or maybe `<h4>`.
- Generally, there should be only one `<h1>` per page. This is the top-level heading (it usually matches or is similar to `<title>` in `<head>`). Just like a book or an article has just one top-level title, so should each page of our site.

Paragraph Tag

- A paragraph, `<p>`, in this context does not necessarily mean a full paragraph like in English class. Instead, it can be any block of text, whether a phrase, sentence, or bona fide paragraph.

Tags for Page Layouts (continued)

Forms

- Generally, a `<form>` tag wraps up a series of `<label>`s and `<input>`s. These can be grouped together for CSS purposes using `<div>`s.
- The most common `<input>` is of type `text`.
- **Every form `<input>` should have an `id` attribute that matches a corresponding `for` attribute on a `<label>`. This is for accessibility purposes.**
- We include a “Submit” button before the closing `</form>`. Optionally, we could also include a `<button type="reset">`.

```
<form>
  <div>
    <label for="first-name">Enter your first name:</label>
    <input type="text" id="first-name" />
  </div>

  <!-- If we don't provide a `type`, this is assumed to be of `type="submit"` -->
  <button>Submit!</button>
  <button type="reset">Reset Form</button>
</form>
```

Tags for Page Layouts (continued)

- Note that for the rest of these examples, `<input>` is shown without `<label>` for brevity.

  ```
  <input type="text" id="first-name" placeholder="First Name"/>
  ```

- If desired, we can also use a `placeholder` attribute to give additional guidance for the `<input>`. **Regardless, we must keep a `<label>` associated.** We do have the option of hiding the `<label>` in an accessible fashion via CSS.

- Following a similar pattern, other `type`s include: `"number"`, `"range"`, `"datetime-local"`, `"search"`, `"color"`, `"checkbox"`, `"radio"`, `"file"`, `"tel"`, `"email"`, `"password"`, etc. A comprehensive list can be found here. Some of these are used more frequently than others.

- Using the correct type of `<input>` allows the browser to help validate some of the `<input>`s without any additional effort from us. For example, `<input type="email" id="email">` will trigger a user error if the user doesn't include an "@" in the field. We should not rely on the browser's validation, however—it is only a small part of form validation practices. We must also include robust JS validation, and have some server-side validation, too.

- For longer forms, we may wish to use `<legend>` and `<fieldset>` to group parts of a form together.

- Another common tag is the `<textarea>`. Just like `<input>`s, we should keep a `<label>`. For this tag, it's not uncommon to specify the attributes `cols` and `rows` to help with rendering. We can also employ more fine-grain control with CSS.

Tags for Page Layouts (continued)

`<a>` vs. `<button>`

- Except for `<form>`s, `<button>`s are used for actions that should be handled by JS. Otherwise, we should use `<a>`. We can use CSS to make our `<a>` look like a button. But, as always, the proper, semantic tag should be used.

Lists

- We can create an **unordered list** via `<ul>`. Inside, we keep our **list items**, `<li>`s.
- Alternatively, we could make an **ordered list** by using `<ol>` instead of `<ul>`.

```
<ul>
  <li>Some bullet point</li>
  <li>Another bullet point</li>
  <li>One more!</li>
</ul>
```

Tags for Page Layouts (continued)

`<nav>`

- We use `<nav>` for a major block of navigation links.
- This would be for a site's primary navigation. Usually, there is only one of these.
- Inside the `<nav>`, we might keep `<ul>`s, `<li>`s, and `<a>`s.

```
<nav>
  <ul>
      <li>
        <!-- If we are already on the page for this link, we can just keep `#` in that one. -->
        <a href="#">Home</a>
      </li>
      <li>
        <!-- It's quite common to include 'relative links.' And, if we structure the site to use
directories with their own 'index.html' file, we can make a cleaner `href` value. -->
        <a href="./about">About Us</a>
      </li>
      <li><a href="./contact">Contact Us</a></li>
  </ul>
</nav>
```

HTML Character Codes

To create special symbols such as the “cents” sign, or even just a simple “non-breaking space,” we use HTML Character Codes. For example, to make a “copyright” symbol, we use `&copy;`.

Time to Code

Review Our Starter Code

Suggested Time:

10 minutes


Break

Time to Code

Write the HTML for a Landing Page

Suggested Time:

30 minutes


Break

CSS

The Role of CSS and Common Theming Techniques, Including Layouts with Flexbox

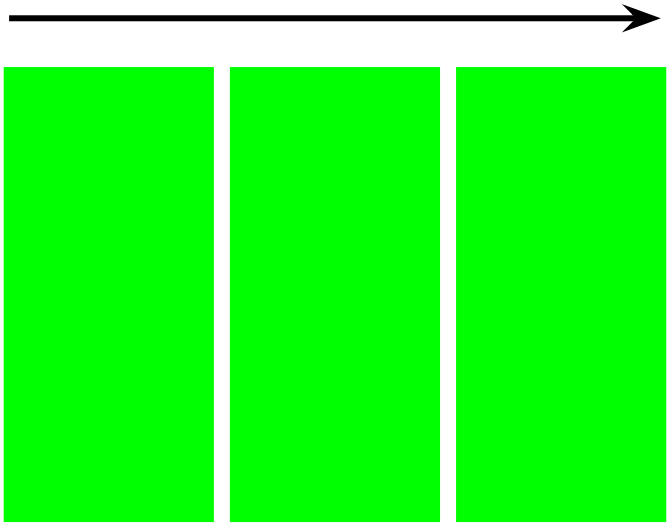
- As opposed to HTML, which just describes our content, CSS applies styles to our content.
- CSS works from the bottom up. The stylesheet loaded last overwrites the previously loaded set of styles in the event that there are overlapping selectors. For this reason, we usually load our custom styles (e.g., `style.css`) last, after any boilerplate CSS resets such as `Normalize.css`.
- `margin` is the spacing **between** elements, while `padding` is the spacing **around** elements.
- `text-align: center` works for basic **text** alignment.
- Applying a specific `width` or `max-width` to a block-level element will allow us to center it with `auto`.

```
div {
  /* The first number, `0` is the top/bottom margin. The `auto` gets applied
for the left/right centering. */
  margin: 0 auto;
  max-width: 550px;
}
```

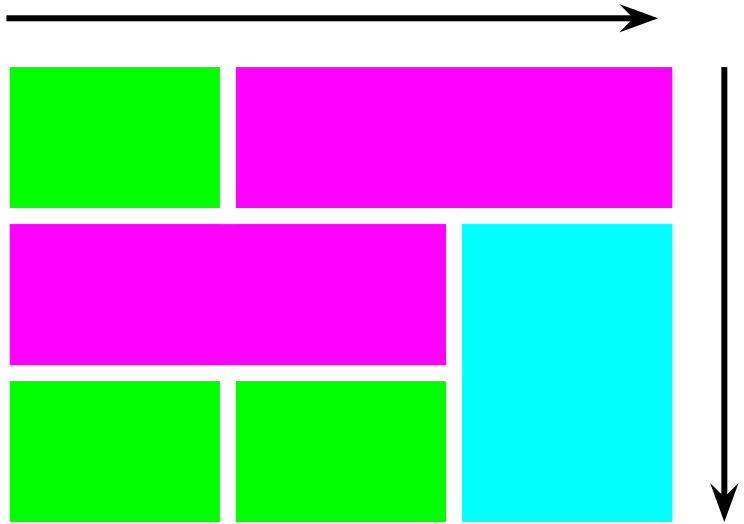
Flexbox

Flexbox has greatly streamlined the ability to quickly align and justify content either horizontally or vertically, and sometimes both.

Flexbox is one dimensional.

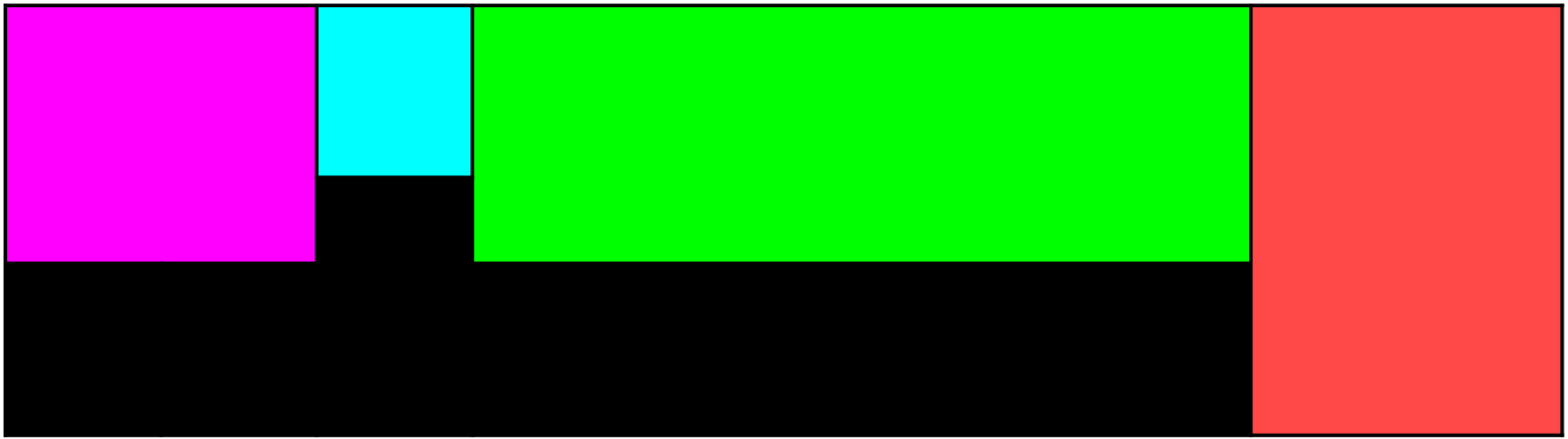


CSS Grids are two dimensional.



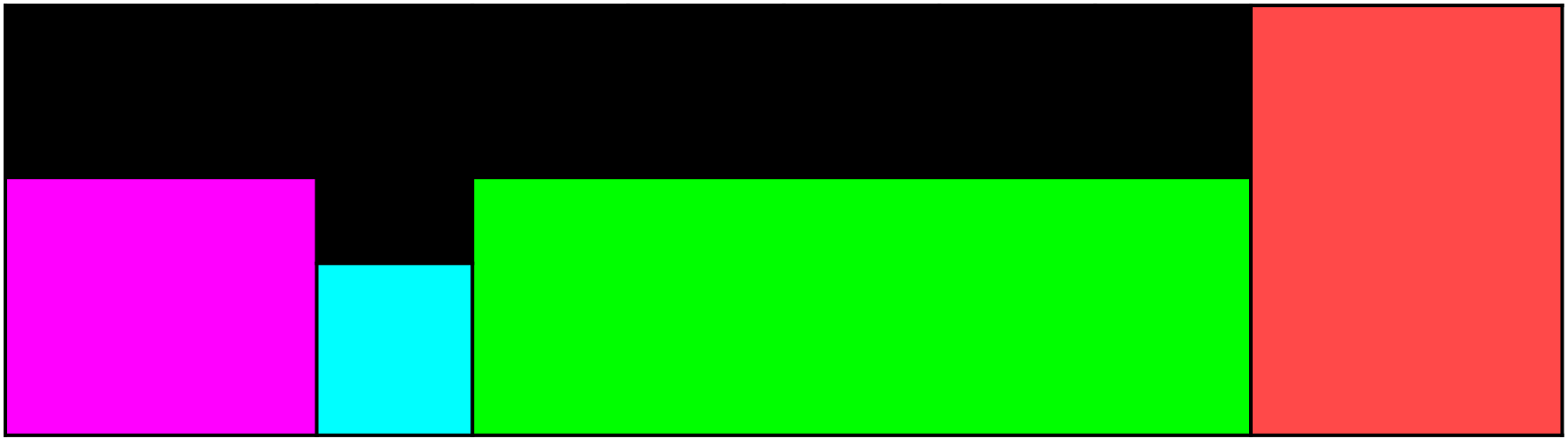
flex-start

```
.flex-container {
    align-items: flex-start;
}
```



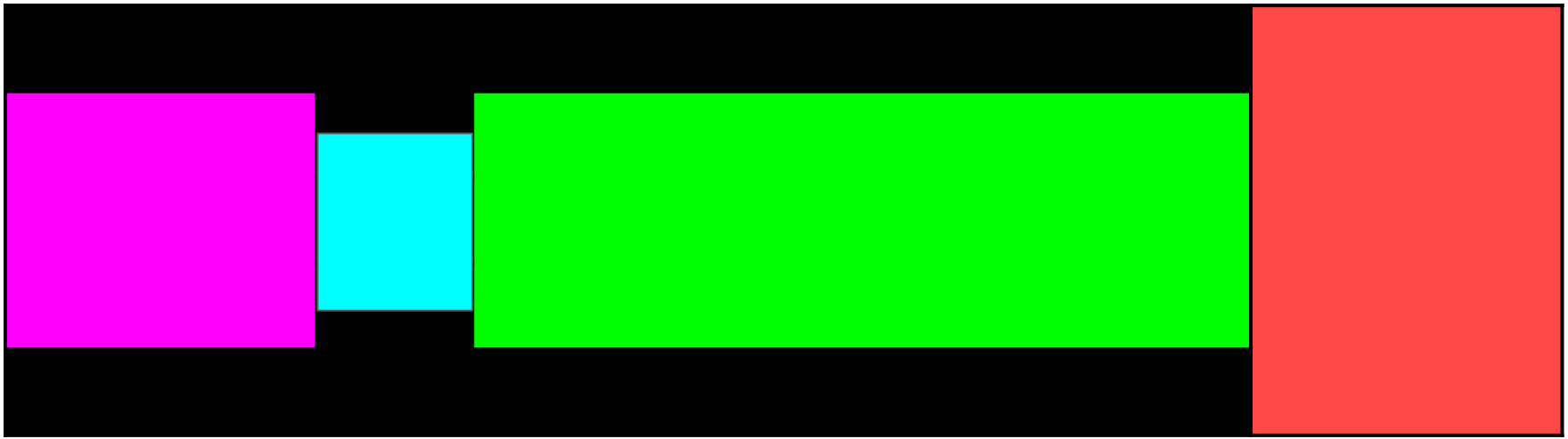
flex-end

```
.flex-container {
    align-items: flex-end;
}
```



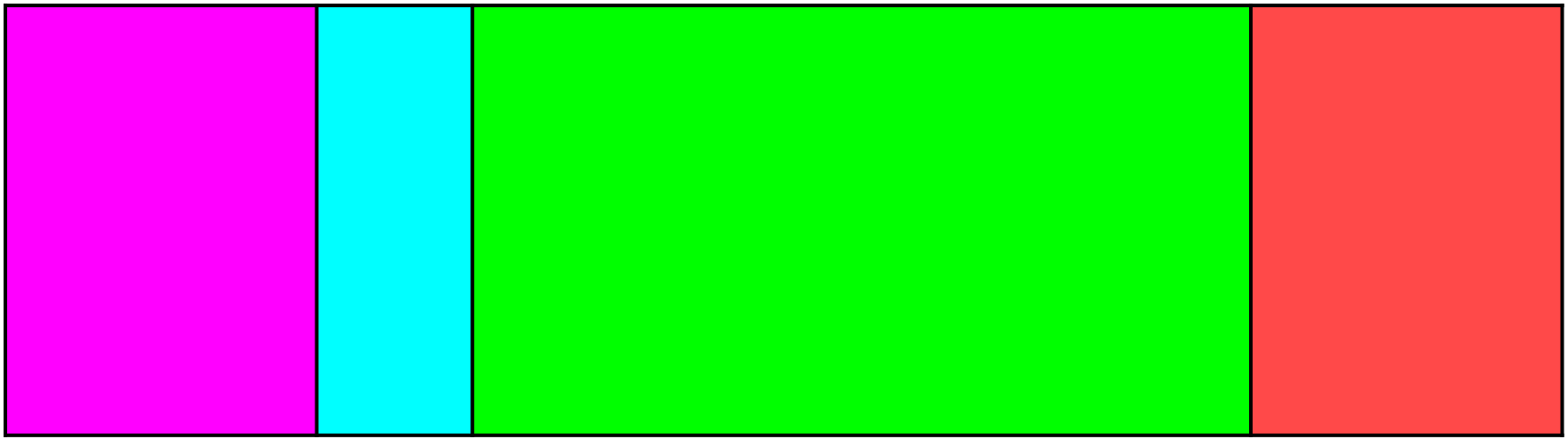
center

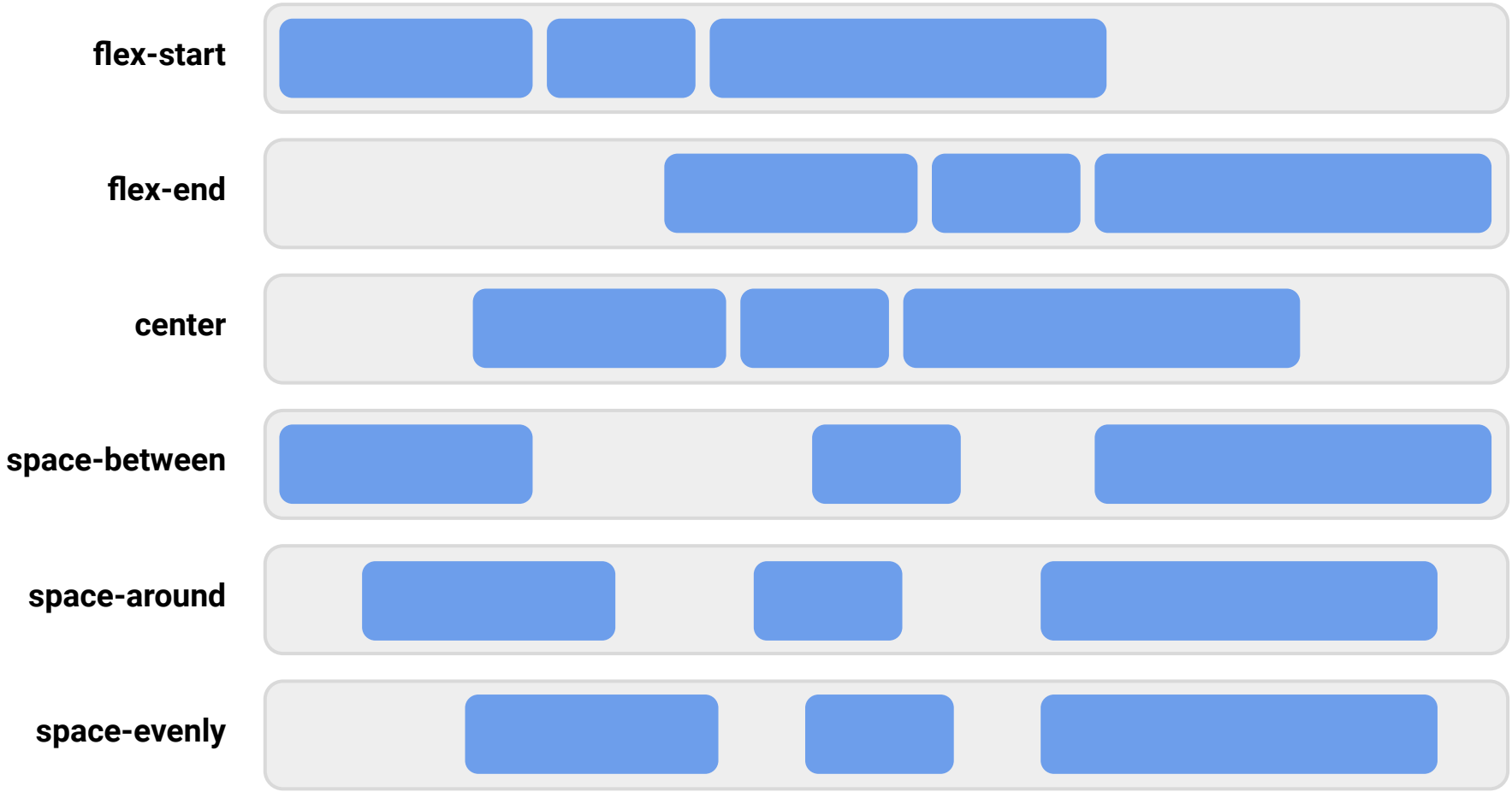
```
.flex-container {
    align-items: center;
}
```



stretch

```
.flex-container {
    align-items: stretch;
}
```




flex-start
flex-end
center
space-between
space-around
space-evenly

CSS Custom Properties (Variables)

- Often, these are applied to the top-level `:root` selector. We need to put `--` followed by a variable name, such as `primary-color`.
- These variables are applied by typing `var` followed by some `()`, with the name of the variable inside of those `()`:

```
:root {
  --some-variable-name: 16px;
}

div {
  /* The margin on all four sides will be 16px. */
  margin: var(--some-variable-name);
}
```

Time to Code

Add CSS Theming and Layout for a Responsive Landing Page

Suggested Time:

30 minutes


Break

Activity: Customize and Complete the Responsive Design

Suggested Time:

30 minutes


RECAP

Questions?