This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan)
(https://twitter.com/davidjmalan)

Lecture 5

- Networking
- HTTP
- HTML
- Forms
- CSS
- JavaScript

Networking

- Today we'll transition from building command-line programs in C to web applications, and though we'll see new languages ideas and concepts will stay the same.
- **TCP/IP** (Transmission Control Protocol and Internet Protocol) are two protocols, or rules that specify how computers can communicate with each other. The modern internet relies on these protocols to work.
- We might have sent handwritten letters in the mail in the past. On the outside of the envelope, we need to write an addres including information like a name, street, and city. We also write our own name and address as the return address.
- Each address, too, should uniquely identify a building or place.
- Our computers also have addresses that uniquely identify them on the internet, called IP addresses. In IPv4, or version 4 of protocol, these addresses are numbers in the format `#.#.#.#`, four numbers between 0 and 255 separated by dots. And to represent each number (with 256 possible values), we need exactly 8 bits, and so each IP address is made of 32 bits. But wi bits, we can only represent 4 billion values. And since there are more than 4 billion devices connected to the internet, we h newer version of the protocol, IPv6, which has 128-bit addresses, that the world is starting to transition to.
- A **server**, which is just a computer connected to the internet that can listen for and respond to messages, might provide ma services, such as a web site or email. To specify that a message is intended for a particular service, such as web browsing, a number called the port number is added to the address. For example, HTTP, for browsing websites, is usually communicated port 80. So an envelope with a message might have `1.2.3.4:80` as the destination address, and `5.6.7.8` as the return And there are other complexities, but that's the basics of how computers can communicate over a network.
- Let's say we wanted to visit a **URL**, Uniform Resource Locator, like `http://www.example.com/`. It turns out that there's an technology called DNS, Domain Name System, that many internet providers and organizations maintain, which converts do names (like `example.com`) into IP addresses.
  - There are actually now hundreds of TLDs, top-level domains in addition to `.com`, such as `.net`, `.org`, `.us`, `.uk`, a more.
  - The `www` in front of a domain name is actually a subdomain, and there might be many of them created, each of which pointing to a different server or set of servers. It's not required, and `www` is only used by convention. For example, MIT `web.mit.edu` for their main website's address.

    - The `/` at the end implies that we're asking for the root page of the site, which is conventionally `index.html`, where indicates that the file is written in HTML, a language we'll soon look at.
- When we type that URL in a browser, our browser first uses DNS to look up the IP address for that domain, and then sends a (in a virtual envelope) to the right IP address for the website. And when the server at that address responds, it will send us content of the website in a virtual envelope with our address as the destination.

HTTP

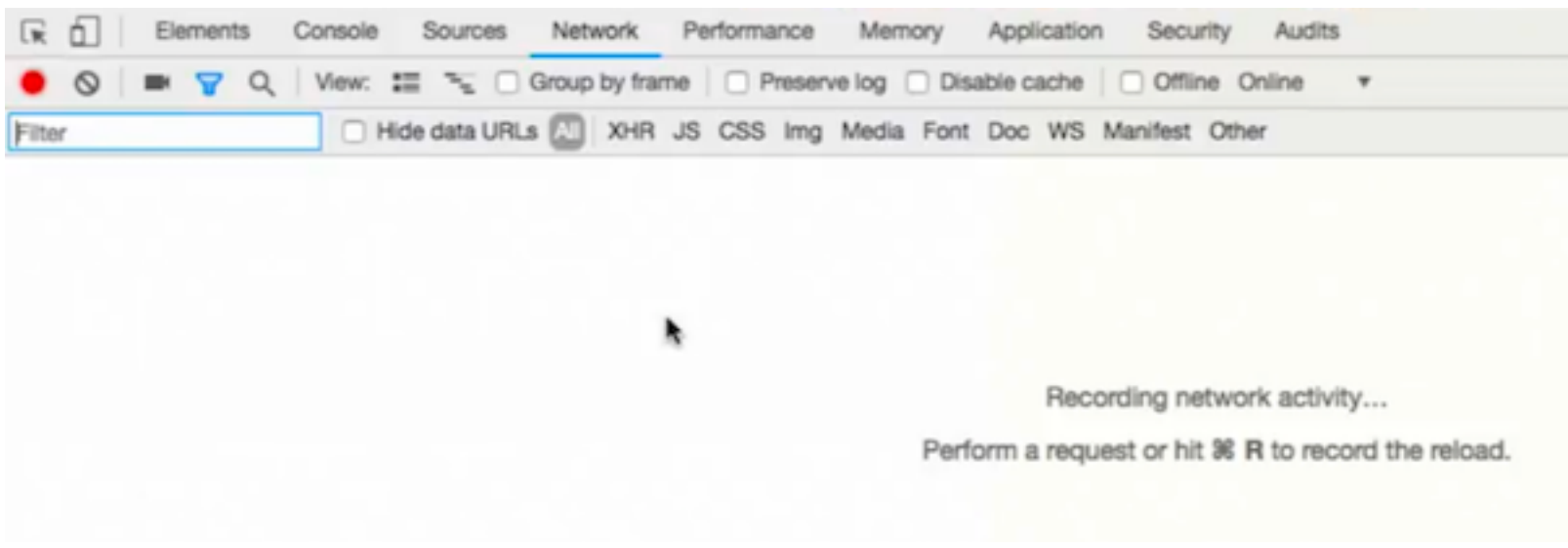
- **HTTP**, Hypertext Transfer Protocol, is another set of rules and conventions for communicating. For example, humans might convention of shaking hands when meeting for the first (or subsequent) times. When our browser communicates to web ser through HTTP, too, both computers follow a protocol for making requests and responses.
- A request for a webpage will look like this:

```
GET / HTTP/1.1
Host: www.example.com
...
```

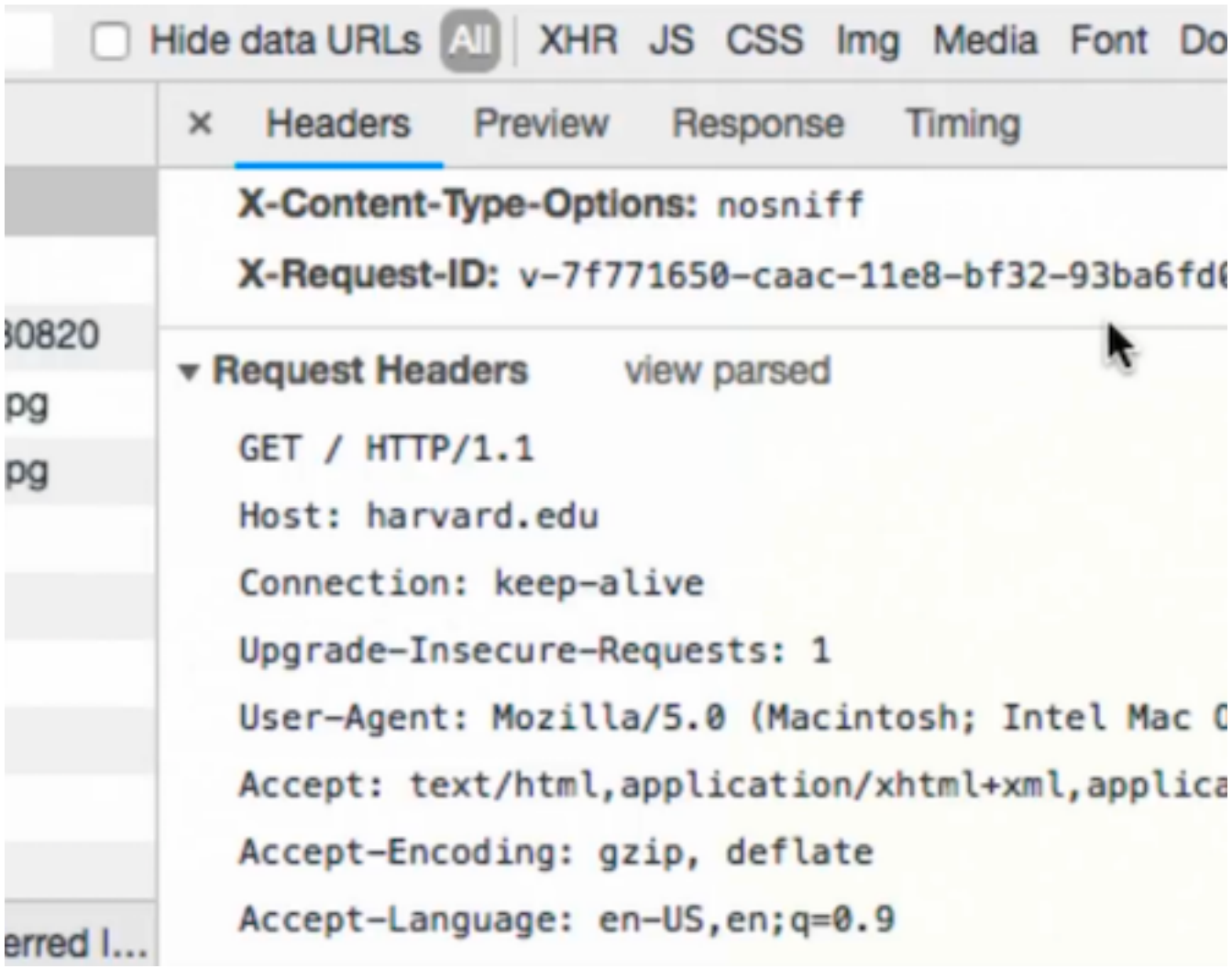
    - `GET` is an HTTP verb that indicates we want to fetch some resource. The `/` indicates we're looking for the default pa `HTTP/1.1` indicates the version of HTTP our browser is using.
    - Then, `Host: www.example.com` is included, since the same server might be listening for and responding to requests multiple websites. There are also other pieces of information included in the `...`, to help the server respond to us appropriately.
- The response from the server might look like this:

```
HTTP/1.1 200 OK
Content-Type: text/html
...
```

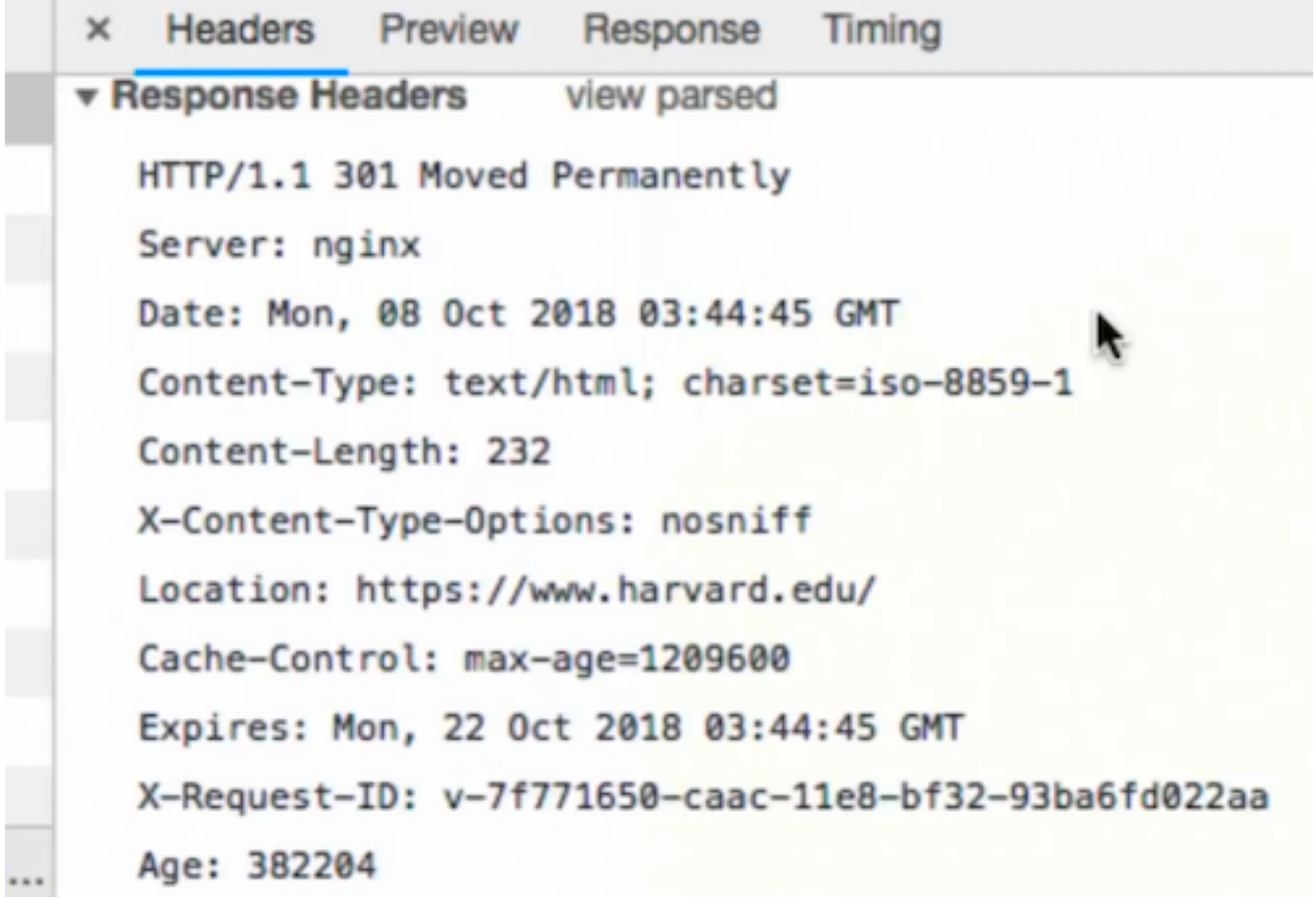
    - First, we get back the version of HTTP, `HTTP/1.1`. Then, `200` is a numeric code that means `OK`, that the server was understand and respond to the request.
    - `Content-Type: text/html` indicates that the content of the response is in the language called HTML, in text forma
- We can open a browser like Chrome, and open the Developer Tools with View > Developer > Developer Tools. A panel will o



- We can click the Network tab, and if we type `harvard.edu` into the address bar and press enter, a lot will happen ver quickly. We can scroll to the very top, click the first request for `harvard.edu`, and see in the right panel, under "Reque Headers", that the browser indeed sends a request that starts with what we expected:



- We can scroll in the same panel and see that the response headers are slightly different:

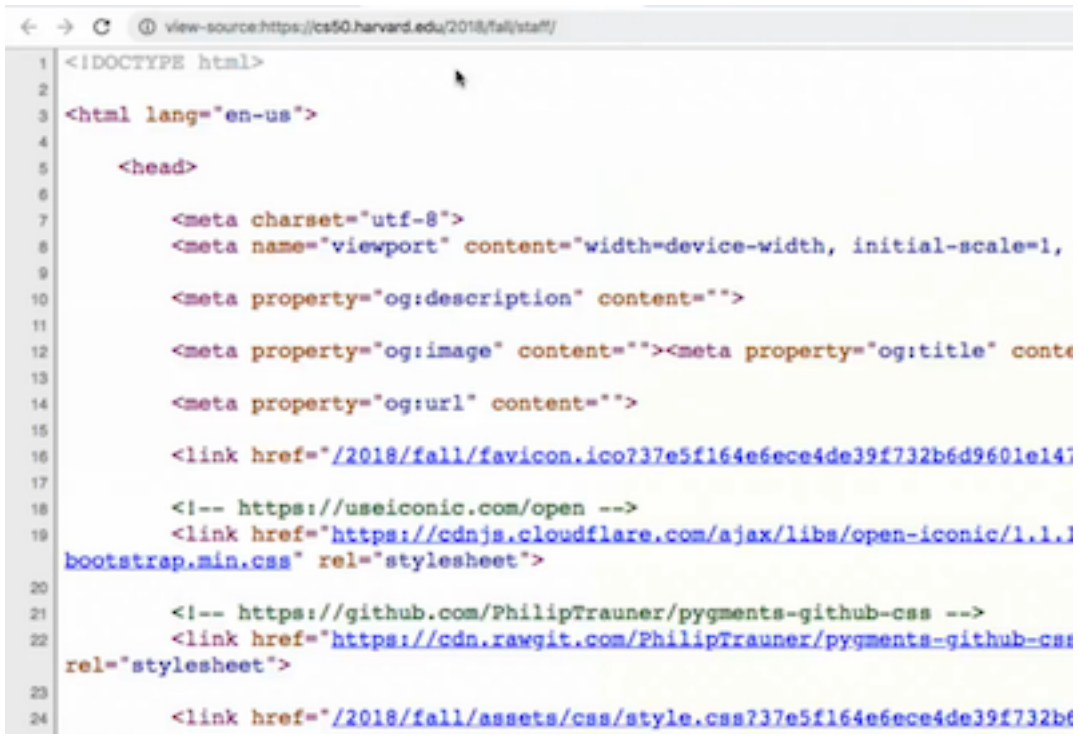


  - The response code, `301`, seems to say "Moved Permanently". And if we look down to "Location:", we see that the new l is `https://www.harvard.edu`. There's a `www`, and also a different protocol, HTTPS, which will encrypt our communi more securely.
- Another HTTP code, `404`, is "Not Found", and we get that back if we're trying to get some URL that the server can't find. Th some interesting ones:
  - `200 OK`
  - `301 Moved Permanently`
  - `302 Found`
  - `304 Not Modified`
  - `401 Unauthorized`
  - `403 Forbidden`
  - `404 Not Found`
  - `418 I'm a Teapot`
  - `500 Internal Server Error`
  - `...`

HTML

- Now that our computers can communicate, we can start thinking about creating the content that websites are comprised of

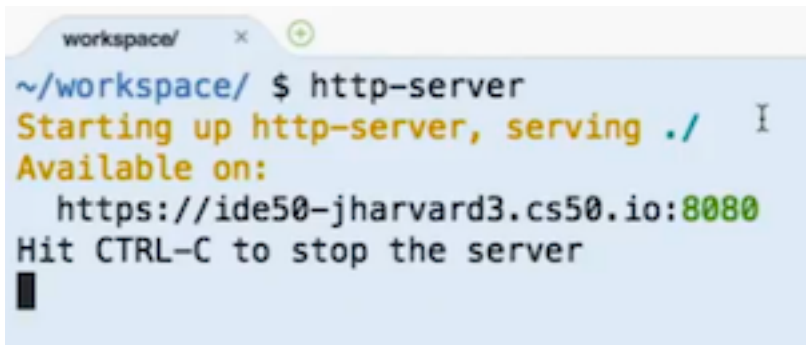
- **HTML**, Hypertext Markup Language, is a standard with which webpages are written. It's interpreted by browsers from top to and each line might have some text, image, or styling instructions.
- In our browser, we can click View > Developer > View Source on a website to see the HTML that drives websites:



  - We can see that this is just text, and the first line, `<!DOCTYPE html>`, indicates to browsers that the page is written in
  - Then, we see a pattern of lines and indentations, and many tags that start with `<` and end with `>`. First, we have the `<html>` tag, and nested inside is a `<head>` tag, which will include information about the webpage, that might not necessarily appear.
  - Then, we eventually see a `<body>` tag, which will have the content of the webpage.
- We can look at a simple example:

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

  - Inside the `<head>` of the webpage, we have a `<title>` tag that indicates the title of our webpage, "hello, title". And have a line with `</title>`, which is a closing tag that indicates the end of the title.
  - Notice that the indentation and opening and closing tags are symmetric. Like in C, the whitespace is not necessary, but stylistically important.
  - The content of this page is just "hello, body".
- With the text editor in CS50 IDE, we can create and save a file called `index.html` with our example code. The CS50 IDE is based, and it can run a web server, which is a program that can listen for and respond to web requests.
- We can run a server in the terminal, called `http-server`, a free and open-source package. If we run that command, we'll s information:



  - `./` is the current directory, and in this case we are in our `~/workspace/` folder.
  - Then, we see a URL to our IDE's web server, and since we want to serve these files separately from the IDE itself, the UI in `:8080`, indicating that we're using port number 8080.
  - If we click that link, we'll see a page that says `Index of /` with the files in our workspace. We can click on `index.h` see our page. We can also change the code in our editor, save, and refresh to see our changes. Since HTML is interprete browser, we don't need to compile it.

- Let's take a look at examples of other tags:

```
<img src="cat.jpg">
```

  - Images can be included with the `<img>` tag, and `src` is an attribute on the tag that modifies it. In this case, it will sp the source of the image, and the value can be a file or other URL. (In the CS50 IDE, we should upload a file called `cat` our workspace folder for this to work.) Finally, we don't close image tags (and other "empty tags"), since there's nothing inside the element.
  - We can also add another attribute tag, `alt`, to add alternative text for the image. So our image will look like this: `<i alt="photo of cat" src="cat.jpg">`
- We can add links with something like `Visit <a href="https://www.harvard.edu/">Harvard</a>.` in our body. The and `Harvard` pieces are just text, but the `<a>` tag surrounding `Harvard` is an anchor tag, which specifies a link with the attribute. In fact, we can phish, or trick, people, into clicking a link to a site that isn't really what they expect. A bad actor cou copy the HTML of some site, and create a site of their own that appears to be the same. (Though, they won't have access to and data stored on the server.)
- We can wrap text with the `<strong>` tag to tell browsers to make it bolder.
- There's also the `<p>` tag for paragraphs:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>paragraphs</title>
    </head>
    <body>
        <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam in tincidunt augue. D
        </p>
        <p>
            Ut tempus rutrum arcu eget condimentum. Morbi elit ipsum, gravida faucibus sodales qu
        </p>
        <p>
            Mauris eget erat arcu. Maecenas ac ante vel ipsum bibendum varius. Nunc tristique nul
        </p>
    </body>
</html>
```

  - Without the `<p>` tags, all of these lines would be displayed together on the page, since HTML ignores whitespace like lines, and instead combines them to at most one space.
- We look at a few more tags from HTML like headings (`<h1>` through `<h6>` indicating the level of heading) and tables (`<tr>` for rows, `<td>` for cells), but through practice and documentation, we can learn to use them fully. Once we understa pattern of tags and attributes, we can write our own HTML.
- We can use tools like the W3C Markup Validator (https://validator.w3.org/) to check that our HTML is valid.

Forms

- On Google, if we search for something, we get redirected to a long URL. It turns out that the URL has our search term in it, a going to a link like `https://www.google.com/search?q=cats` will bring us directly to the results page for a search for "
  - The page is called `search`, and that goes to code on their servers that generates a response for that page dynamicall programmatically.
  - The `?` in the URL adds additional input for the page, and `q=cats` is telling the server that we are passing in "cats" fo input (search box in this case) with the name "q", which probably stands for "query".
- We can write the HTML for a form that takes us to the Google search results for some user input:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>search</title>
    </head>
    <body>
        <form action="https://www.google.com/search" method="get">
            <input name="q" type="text">
            <input type="submit" value="Search">
        </form>
    </body>
</html>
```

- With the `form` tag, we can create a form. The `action` attribute tells the browser where the form should go, and the `method` attribute indicates how to send the form inputs.
- The first `input` tag is a text box, which we will name `q` so that it can be sent to Google correctly, and the second `i` tag is a submit button that we'll label "Search".

CSS

- While HTML is used for layout and structure, **CSS**, Cascading Style Sheets, is another language we can use to style, or chang aesthetics, of our webpages.
- Let's take a look at `css0.html`:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>css0</title>
    </head>
    <body>
        <header style="font-size: large; text-align: center;">
            John Harvard
        </header>
        <main style="font-size: medium; text-align: center;">
            Welcome to my home page!
        </main>
        <footer style="font-size: small; text-align: center;">
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

  - Here, for each of these tags, we've added a `style` attribute and some set of key-value pairs as the value that will app just those elements. These pairs, like `font-size: large;`, are setting CSS properties and can change many aesthetic aspects of elements.
  - Notice that we have semantic, or meaningful, tags like `<header>`, `<main>`, and `<footer>` that separates our page i sections.
- Since CSS is inherited by nested elements in HTML, we can factor out the common styles:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>css1</title>
    </head>
    <body style="text-align: center;">
        <header style="font-size: large;">
            John Harvard
        </header>
        <main style="font-size: medium;">
            Welcome to my home page!
        </main>
        <footer style="font-size: small;">
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

- Here, the `text-align: center;` style is applied to the `<body>` element, so it will cascade, or be inherited by each inside `<body>`.
- We can factor out CSS into the `<head>`, with CSS classes:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <style>

            .centered
            {
                text-align: center;
            }

            .large
            {
                font-size: large;
            }

            .medium
            {
                font-size: medium;
            }

            .small
            {
                font-size: small;
            }

        </style>
        <title>css2</title>
    </head>
    <body class="centered">
        <header class="large">
            John Harvard
        </header>
        <main class="medium">
            Welcome to my home page!
        </main>
        <footer class="small">
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

- Now, the HTML in the `<body>` specifies a `class` for each element, but all the CSS for the styling has been moved to `<head>`, so we can compartmentalize it more easily. And in CSS, we use `.something` to apply properties to element class of `something`. Each class, too, can have many CSS properties, not just one.
- We could even apply CSS to all elements of a certain type, using CSS selectors:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <style>

            body
            {
                text-align: center;
            }

            header
            {
                font-size: large;
            }

            main
            {
                font-size: medium;
            }

            footer
            {
                font-size: small;
            }

        </style>
        <title>css3</title>
    </head>
    <body>
        <header>
            John Harvard
        </header>
        <main>
            Welcome to my home page!
        </main>
        <footer>
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

- Notice that now we can use `body` and `header` to select those elements, without attaching a class to them in the HT
- Finally, we can include external stylesheets, or CSS in separate files, that multiple HTML pages can include and share:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <link href="css4.css" rel="stylesheet">
        <title>css4</title>
    </head>
    <body>
        <header>
            John Harvard
        </header>
        <main>
            Welcome to my home page!
        </main>
        <footer>
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

- We need to create a file called `css4.css`, and place our CSS code inside that, for this to work. But now we can use th `<link>` tag to include it.
    - There are tradeoffs, too, to having separated CSS files, since a simple webpage may not need the additional complexity overhead of a linked stylesheet. But having separation of concerns allows for easier collaboration and clearer organiza code.
- Phew, we covered lots of concepts here! But, now that we're familiar with some of these patterns, we can learn to use addit features by reading examples and documentation online.

JavaScript

- **JavaScript**, a programming language, can be used on our webpages to make them more dynamic. The user's browser runs th JavaScript code we write, to make changes to the page.
- JavaScript is similar to C, and is interpreted by a browser from top to bottom.
- Many of the programming elements are the same:

```
let counter = 0;
```

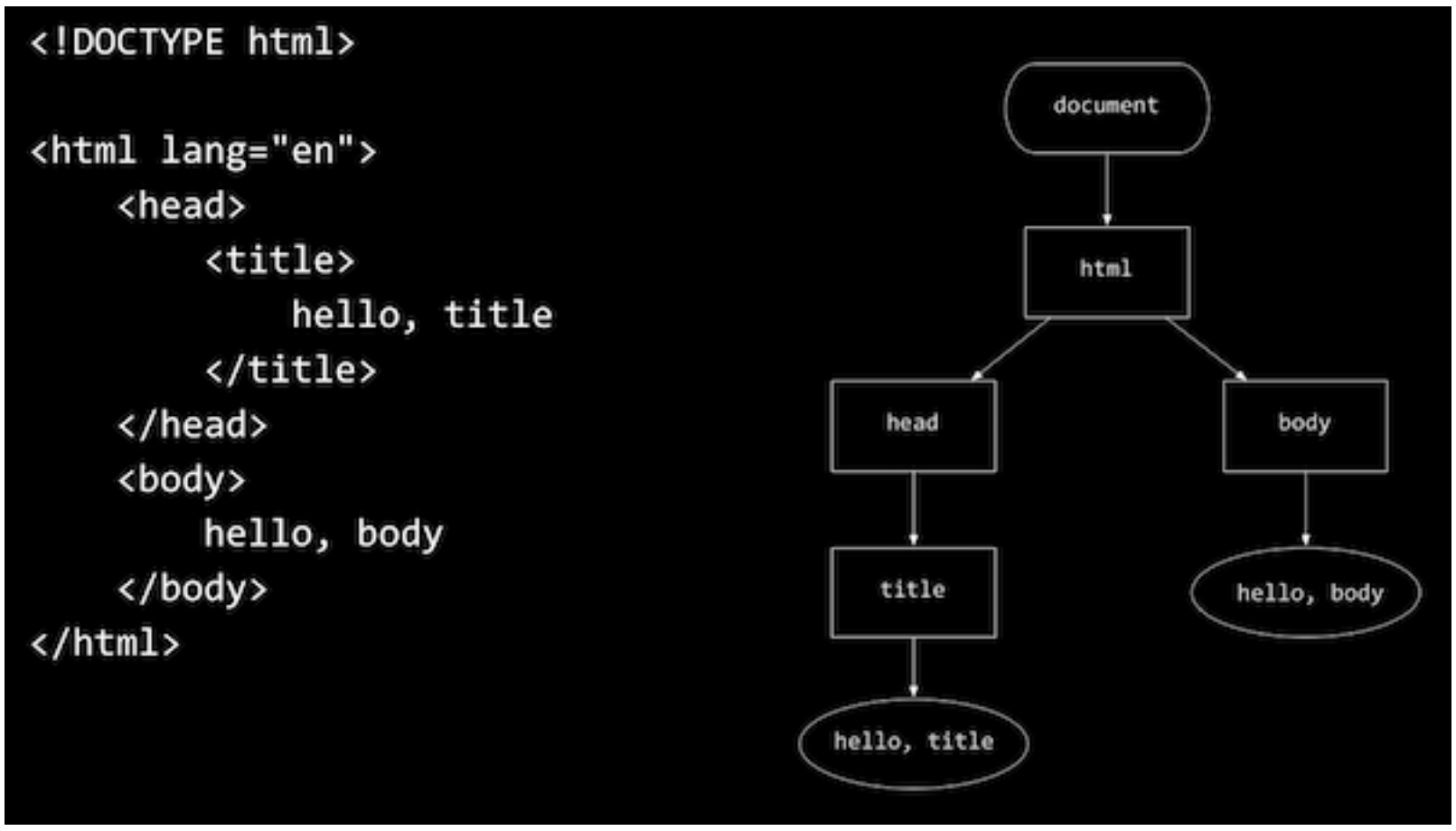
    - We use the `let` keyword in JavaScript to initialize a variable, and we don't need to specify what the type of the variab be.
- Adding 1 to a variable has the exact same syntax as it does in C.

```
counter = counter + 1;
counter += 1;
counter++;
```

- Conditions and loops, too, are the same.

```
if (x < y)
{

}
else if {

}
else
{

}

while (true)
{

}

for (let i = 0; i < 50; i++)
{

}
```

- Our example webpage can be represented by a tree, in what's called the DOM, Document Object Model:



  - Notice that each node is an element on the page, and nested nodes show as children nodes. A browser, when it loads a webpage, automatically builds a tree in memory with elements from the HTML.
- With JavaScript, we can add or change any of these nodes in the DOM.
- We can make an interactive page like the following:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <script>

            function greet()
            {
                alert('hello, ' + document.querySelector('#name').value);
            }

        </script>
        <title>hello1</title>
    </head>
    <body>
        <form onsubmit="greet(); return false;">
            <input autocomplete="off" autofocus id="name" placeholder="Name" type="text">
            <input type="submit">
        </form>
    </body>
</html>
```

- We have a form element in the `<body>` with a text input and a submit button. But when the form is submitted, we wa
  browser to call a `greet()` function, and with `return false;`, we tell the browser to do nothing else with the form
  put that into the `onsubmit` attribute of the form. Notice that we also have `id="name"` for the text input element. Th
  `autocomplete="off"` attribute turns off the autocomplete in the browser, and `autofocus` selects the input box wh
  page is loaded so the user can start typing into it right away.
- The `greet()` function is defined in the `<head>` of our page, inside a `<script>` tag that allows us to write our own
  JavaScript. In JavaScript, we can define a function with the `function` keyword, and if it takes no inputs, we can simply
  `()`. And this function in turns calls the `alert()` function, which is built into browsers, to create an alert box.
- The content of the alert box will be `hello,` plus the value of the element in the webpage (called `document`) with t
  `name`. The `querySelector` function is attached to the object that represents the webpage, so we call it with
  `document.querySelector()`. Then, the element that gets selected will also has an attribute called `value` that we
  access with `.value`.

- We look at another example, that can change the style of a webpage:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>background</title>
    </head>
    <body>
        <button id="red">R</button>
        <button id="green">G</button>
        <button id="blue">B</button>
        <script>

            let body = document.querySelector('body');
            document.querySelector('#red').onclick = function() {
                body.style.backgroundColor = 'red';
            };
            document.querySelector('#green').onclick = function() {
                body.style.backgroundColor = 'green';
            };
            document.querySelector('#blue').onclick = function() {
                body.style.backgroundColor = 'blue';
            };

        </script>
    </body>
</html>
```

- It turns out that we can attach JavaScript functions to events in the browser, like the following:
    - `blur`
    - `change`
    - `click`
    - `drag`
    - `focus`
    - `keypress`
    - `load`
    - `mousedown`
    - `mouseover`
    - `mouseup`
    - `submit`
    - `touchmove`
    - `unload`
    - `...`
- We can add code called event listeners to elements like `document.querySelector('#red')`. The `onclick` value of eac element can be a function that is automatically called by the browser, when the element is clicked. And the function attach doesn't have a name, but is defined with `function() {}`.
- With `body.style.backgroundColor`, we can access the `style` of the `body`, and set its `backgroundColor` value.
- We can change the font size, too:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>size</title>
    </head>
    <body>
        <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam in tincidunt augue. D
        </p>
        <select>
            <option value="xx-large">xx-large</option>
            <option value="x-large">x-large</option>
            <option value="large">large</option>
            <option selected value="initial">initial</option>
            <option value="small">small</option>
            <option value="x-small">x-small</option>
            <option value="xx-small">xx-small</option>
        </select>
        <script>

            document.querySelector('select').onchange = function() {
                document.querySelector('body').style.fontSize = this.value;
            };

        </script>
    </body>
</html>
```

- We have a set of `option` elements in a `select` (a dropdown menu in HTML that we can look up the documentation and now, whenever the `select` element is changed, we set the `fontSize` of the `style` of the `body` element. We value to `this.value`, and `this` refers to the `select` element when the function is called, since the function is ca from that element.
- We can write a page with an element that blinks, or appears and disappears repeatedly:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <script>

            // Toggles visibility of greeting
            function blink()
            {
                let body = document.querySelector('body');
                if (body.style.visibility == 'hidden')
                {
                    body.style.visibility = 'visible';
                }
                else
                {
                    body.style.visibility = 'hidden';
                }
            }

            // Blink every 500ms
            window.setInterval(blink, 500);

        </script>
        <title>blink</title>
    </head>
    <body>
        hello, world
    </body>
</html>
```

- We use the `visibility` attribute to make the `body` visible or hidden, and `window.setInterval` to call this funct every 500 milliseconds.

- Browsers also have a geolocation function, which we can call to get the user's current location:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>geolocation</title>
    </head>
    <body>
        <script>

            navigator.geolocation.getCurrentPosition(function(position) {
                document.write(position.coords.latitude + ", " + position.coords.longitude);
            });

        </script>
    </body>
</html>
```

- `navigator` refers to the user's browser, and the `geolocation.getCurrentPosition` function will return a `posit` object. When we get that `position` object, we want to call a function that will then write the `latitude` and `longi` values to the document.