Lession 1

Dynamic versus static code

The word **dynamic** is used to describe both client-side JavaScript, and server-side languages — it refers to the ability to update the display of a web page/app to show different things in different circumstances, generating new content as required. Server-side code dynamically generates new content on the server, e.g. pulling data from a database, whereas client-side JavaScript dynamically generates new content inside the browser on the client, e.g. creating a new HTML table, filling it with data requested from the server, then displaying the table in a web page shown to the user. The meaning is slightly different in the two contexts, but related, and both approaches (server-side and client-side) usually work together.

A web page with no dynamically updating content is referred to as **static** — it just shows the same content all the time.

How do you add JavaScript to your page?

JavaScript is applied to your HTML page in a similar manner to CSS. Whereas CSS uses <link> elements to apply external stylesheets and <style> elements to apply internal stylesheets to HTML, JavaScript only needs one friend in the world of HTML — the <script> element. Let's learn how this works.

Internal JavaScript

1. First of all, make a local copy of our example file apply-javascript.html. Save it in a directory somewhere sensible.
2. Open the file in your web browser and in your text editor. You'll see that the HTML creates a simple web page containing a clickable button.
3. Next, go to your text editor and add the following in your head — just before your closing </head> tag:

   HTMLCopy to Clipboard
   ```
   <script>
    // JavaScript goes here
   </script>
   ```

4. Now we'll add some JavaScript inside our <script> element to make the page do something more interesting — add the following code just below the "// JavaScript goes here" line:

   JSCopy to Clipboard
   ```
   document.addEventListener("DOMContentLoaded", () => {
    function createParagraph() {
     const para = document.createElement("p");
     para.textContent = "You clicked the button!";
     document.body.appendChild(para);
   ```

```
}

const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", createParagraph);
}
});
```

5. Save your file and refresh the browser — now you should see that when you click the button, a new paragraph is generated and placed below.

Note: If your example doesn't seem to work, go through the steps again and check that you did everything right. Did you save your local copy of the starting code as a .html file? Did you add your <script> element just before the </head> tag? Did you enter the JavaScript exactly as shown? **JavaScript is case sensitive, and very fussy, so you need to enter the syntax exactly as shown, otherwise it may not work.**

Note: You can see this version on GitHub as apply-javascript-internal.html (see it live too).

External JavaScript

This works great, but what if we wanted to put our JavaScript in an external file? Let's explore this now.

1. First, create a new file in the same directory as your sample HTML file. Call it script.js — make sure it has that .js filename extension, as that's how it is recognized as JavaScript.
2. Replace your current <script> element with the following:

   HTMLCopy to Clipboard
   ```
   <script src="script.js" defer></script>
   ```

3. Inside script.js, add the following script:

   JSCopy to Clipboard
   ```
   function createParagraph() {
     const para = document.createElement("p");
     para.textContent = "You clicked the button!";
     document.body.appendChild(para);
   }

   const buttons = document.querySelectorAll("button");

   for (const button of buttons) {
     button.addEventListener("click", createParagraph);
   }
   ```

4. Save and refresh your browser, and you should see the same thing! It works just the same, but now we've got our JavaScript in an external file. This is generally a good thing

in terms of organizing your code and making it reusable across multiple HTML files. Plus, the HTML is easier to read without huge chunks of script dumped in it.

Note: You can see this version on GitHub as apply-javascript-external.html and script.js (see it live too).

Inline JavaScript handlers

Note that sometimes you'll come across bits of actual JavaScript code living inside HTML. It might look something like this:

JSPlayCopy to Clipboard
```
function createParagraph() {
  const para = document.createElement("p");
  para.textContent = "You clicked the button!";
  document.body.appendChild(para);
}
```
HTMLPlayCopy to Clipboard
```
<button onclick="createParagraph()">Click me!</button>
```

You can try this version of our demo below.

Play

This demo has exactly the same functionality as in the previous two sections, except that the <button> element includes an inline onclick handler to make the function run when the button is pressed.

Please don't do this, however. It is bad practice to pollute your HTML with JavaScript, and it is inefficient — you'd have to include the onclick="createParagraph()" attribute on every button you want the JavaScript to apply to.

Using addEventListener instead

Instead of including JavaScript in your HTML, use a pure JavaScript construct.
The querySelectorAll() function allows you to select all the buttons on a page. You can then loop through the buttons, assigning a handler for each using addEventListener(). The code for this is shown below:

JSCopy to Clipboard
```
const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", createParagraph);
}
```

This might be a bit longer than the onclick attribute, but it will work for all buttons — no matter how many are on the page, nor how many are added or removed. The JavaScript does not need to be changed.

Note: Try editing your version of apply-javascript.html and add a few more buttons into the file. When you reload, you should find that all of the buttons when clicked will create a paragraph. Neat, huh?

Script loading strategies

There are a number of issues involved with getting scripts to load at the right time. Nothing is as simple as it seems! A common problem is that all the HTML on a page is loaded in the order in which it appears. If you are using JavaScript to manipulate elements on the page (or more accurately, the Document Object Model), your code won't work if the JavaScript is loaded and parsed before the HTML you are trying to do something to.