Browsers and the Front-end

HTML

Tree Structure

Designed for documents accessed remotely, intended to be used by browsers - sort of like "SGML on wheels". They are both the same in that they are an ***attributed*** *tree of text*.

The nodes in the tree have **attributes**. The leaves of the tree are text strings.

```
<p align="right">Hello</p>
<!-- ^ name of the ATTRIBUTE -->
<!--        ^ value of the ATTRIBUTE -->
<!--               ^ text CONTENT of the node -->
```

The HTML tree is an example of **serialization** - a standard way to convert a data structure into a string of bytes. A standard should also specify how to **deserialize** the bytes back into the original data structure.

Terminology

An **element** is a node in the tree. This whole thing is an element:

```
<head><!-- things here --></head>
```

The words enclosed in angle brackets are **tags**:

```
<head><!-- things here --></head>
<!-- ^ opening tag           ^ closing tag-->
```

A **self-closing element** does not have any content, and it can be abbreviated like so:

```
<br />
```

A **void element** is an element that never has any subtrees. These do not have to have a slash in the tag:

```
<meta charset="UTF-8">
```

A **raw text element** is an element that can contain only text:

```
<verbatim>lorem ipsum</verbatim>
```

A **normal element** can have sub-elements:

```
<div>
  <p>hello there</p>
</div>
```

DTD (Document Type Declaration)

How do you know the context of an element's syntax?

Some points of concern regarding the syntax of an element include:

- void, normal, raw text?
- what attributes are allowed?
- what restrictions on sub-elements?

This context is supplied by a **Document Type Definition (DTD)**, an idea inherited from SGML.

The Problem with DTD:

The rapid evolution of browsers prompts the continued creation of new DTDs for new classes of browsers.

The client-server model makes this a mess because the client and server need to agree on the interface between the two. DTDs need to be contracts/protocols for *interoperability* between the browser writer and the server writer.

Historically, there grew versions 1 to 4 that followed the **Internet RFC**. This process eventually broke down because there was too much bureaucracy in putting out a new DTD version.

HTML5

A new, *evolving* standard was created, **HTML5**. This standard is edited sporadically/constantly overseen by a committee.

HTML5 does not just cover text. It also covers the intended meaning of the element, not just the layout/presentation.

It's as if it's written:

```
<p> ... </p>
"This denotes a paragraph."
```

The exact same HTML document can appear differently on different screens depending on how the browser chooses to **render** it. This is advantageous because it allows pages to have **responsive** format depending on the device they are rendered on. HTML5 separates the presentation from the styling.

> **ASIDE:** Many big companies have an incentive to join the consortium overseeing the development of projects like HTML5 because that gives them say in what happens and prevents a single entity from changing things in a direction that only benefits them.

DOM (Document Object Model)

HTML5 also specifies the **Document Object Model (DOM)**, the standard way to access the tree representing HTML in an object-oriented program. It is like the API for tree manipulation.

The DOM unifies the stack:

- It is most often it is used inside the browser. Code inside the browser can look inside the tree.
- The server can also use the DOM to figure out what tree to build before serializing it and sending it to the browser.
- Intervening networks like routers between the client and server can also use the DOM.

JS (JavaScript)

By default, there is a DOM **binding** for JavaScript, meaning you can traverse the tree within the programming language.

PROBLEM: JavaScript is a pain because it is *too powerful*, besides being an annoying language in general (very not biased). Any code can be put into a browser and introduce bugs, portability issues, etc. The debugging becomes difficult. This led to the introduction of CSS.

> **HISTORICALLY:** JavaScript is simple and interpreted, and was chosen primarily because it happened to be a suitable programming language during a time where one was much needed to support the growing Internet. It was originally written in about a week. Had Python or another language been ready and available at the time, it may be a different story today.

In the DOM

JavaScript's key notion is that it is hooked into HTML. JavaScript can be written directly in the DOM:

```
<script>console.log("hello world")</script>
<script src="myscript.js"></script>
```

The `src` attribute specifies the file to fetch. If it does not begin with a protocol (e.g. `http`/`https`), then it is resolved as a file local to the website domain.

The code within the `<script>` element has full access to the DOM, including:

- Examining the DOM
- Modifying the DOM
- Performing actions (e.g. `alert`)

This entails writing a lot of code, which makes it error-prone. Thus, **JSX** was created as a way to simplify the generation of DOM from JavaScript code.

JSX (JavaScript Extension)

Used in frameworks like React.

Uses a **syntax extension** to JavaScript. JSX code needs to be mechanically transformed to vanilla JavaScript before it can be interpreted.

This is a common pattern in software called **preprocessing**, where code is first written in a more understood fashion before being converted to its functional equivalent.

An angle bracket `<` at the start of an expression is invalid in vanilla JavaScript, so `<expression>` denotes the departure from normal JavaScript and the start of a JSX expression:

```
const language = "en";
const class_ = "CS 35L";
const header = (
  <h1 lang={language}>
    {class_} assignment {n+1}
  </h1>
);
```

Likewise, curly braces `{}` are invalid in vanilla HTML, so encountering an expression like `{expression}` denotes an embedded return to the JavaScript world. The expression enclosed in the braces can be any valid JavaScript, with the same access to the file namespace and DOM like vanilla JavaScript.

CSS (Cascading Style Sheets)

Designed as a **declarative** spec for what appears on the screen. **Declarative** means we specify what we want, not how to get it. Generally designed more for the **web designers** on a team than the **developers**.

CSS is a compromise between the presentational layer of HTML and the interactive layer of JavaScript. With these three languages we now have the modern **separation of concerns**:

- **HTML:** Presentation (Structure)
- **CSS:** Styling (Appearance)
- **JavaScript:** Interactivity (Functionality)

CSS statements are put into DOM style elements. The styles are *cascading*, meaning by default they are inherited by subtrees.

Competing Style Sources

There are competing sources for putting styles into the DOM tree:

1. Author of the webpage.
2. The user of the browser.
3. The browser configuration (phone, laptop, etc.).

CSS combines these sources by specifying rules for combination. In general, it is hierarchical, with the author's styles overriding the user's, which overrides the browser's, etc.

Web Browsers

Browser Rendering Pipeline

Abstract model:

```
HTML doc --> Browser[..., parse, ..., execute, ...] --> screen pixels
```

The browser will:

1. Parse the incoming HTML document.
2. Build up the DOM.
3. **Render** (turn into pixels) the DOM onto the screen.

By default, this is very slow, so browser writers employ various techniques to make rendering as fast as possible.

Rendering Right Away

One solution is to make the browser start rendering before having complete information about the webpage.

In practice the HTML document is received in packets, so the browser can already read part of the document while the rest is still incoming. The root tends to arrive first, which can be parsed right away to load metadata the browser can use to deduce many details about the document.

Optimization Techniques

1. The browser can guess whether an element will be rendered onto the screen (as opposed to being hidden at the bottom of the page outside the viewport). If not, skip it for now.

> This means that as a JavaScript developer, you cannot assume that code will always be executed immediately because the part of the document containing the script might not be loaded yet if ever.

2. The browser may also decide that some elements are low priority, in which case, defer execution.
3. The browser can guess the overall layout of an element (like a `<table>`) and render it based on its guess. If the guess is wrong, re-render.

Data Interchange Formats

These are standards to represent a tree structure in a compact format suitable for transmitting over networks.

JSON (JavaScript Object Notation)

The standard way to represent a tree structure over the Internet.

EXAMPLE: The JSON representation of a popup menu:

```
```

```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuItems": [
        {
          "value": "New",
          "onClick": "CreateNewDoc()"
        },
        {
          "value": "Close",
          "onClick": "CloseDoc()"
        }
      ]
    }
  }
}
```

XML (Extensible Markup Language)

Like HTML but designed for data.

EXAMPLE: The same popup menu from the JSON example:

```
<menu id="file" value="file">
  <popup>
    <menuitem value="New" onClick="CreateNewDoc()">
    <menuitem value="click" onClick="CloseDoc()">
  </popup>
</menu>
```

Different network protocols can use different formats.

In general, JSON tends to be more popular. With JavaScript code, it's easier to parse strings in JSON fields than in XML attributes because JSON by design is already valid JavaScript. JSON is also slightly smaller in terms of file sizes.