HTML, the DOM, CSS, JavaScript, JSX, and Python Dictionaries

1. HTML: Structure and Design Principles

1.1 Origins and Goals

- HTML is based on SGML (Standard Generalized Markup Language).
- SGML was designed with two key ideas:
  - **Portability** across different publishers.
  - **Separation** of content from form (presentation).
- HTML inherits the idea that:
  - **Content**: What the user wants to convey.
  - **Form**: How content is presented (appearance/style).
- HTML is declarative: it specifies desired output, not the steps required to achieve it.

1.2 Declarative vs. Imperative

- HTML is a declarative language: it describes what should appear on the screen (structure + content).
- JavaScript is imperative: describes how things should be done procedurally(logic + behavior).
- Original design philosophy:
  - Prefer HTML and CSS for content and styling.
  - Use JavaScript only when needed to control behavior.
- This philosophy wasn't always maintained in practice.

1.3 Goals in Designing HTML

- Support consistent rendering across different devices.
- Ensure that the same content can appear appropriately on laptops, phones, etc.
- Allow browsers the flexibility in rendering to improve usability and accessibility.

1.4 HTML Syntax and Elements

1.4.1 HTML Elements

- An HTML element corresponds to a node in the document's tree structure (DOM).
- Basic syntax:

```
<tagname attribute="value">Content</tagname>
```

- The tree is kept in memory (DOM) by browsers and manipulated via JavaScript.

1.4.2 Nesting and Valid Syntax

- Tags must be properly nested.
  - Invalid:

```
<a><b>…</a></b>
```

  - Valid:

```
<a><b>…</b></a>
```

- Mismatched or improperly nested tags typically render incorrectly, but browsers are forgiving.

1.4.3 Optional and Void Tags

- Closing tags can sometimes be omitted if the structure is unambiguous.
- Void elements: represent standalone elements with no children.

```
<br>
```

- Equivalent to:

```
<br></br>
```

But HTML recommends using the shorter form.

1.5 Attributes

- Elements can have attributes: name-value pairs.
- Syntax:

```
<img src="image.jpg" width="500" height="300">
```

- The attribute name and value are strings and must be unique per element.
- Attributes must appear in the opening tag.

1.6 Types of Elements

Element Type	Characteristics	Example Tags
Normal	Can contain mixed content and children	`<p>`, `<div>`
Raw Text	Can only contain text (rare)	Could be `<script>`
Void	Cannot have any content or children	` `

1.7 DTD and HTML Versions

- Early HTML specified strict rules using DTD (Document Type Declaration).
- DTD specifies:
  - Which elements are allowed.
  - Their types (void, raw, normal).
  - Allowed attributes.
  - Valid nesting and element hierarchy.
- HTML 1–4 used this system.
- Issues:
  - Too rigid and slow to adapt to evolving browser innovation.
  - Standards couldn't keep pace with browser capabilities.

1.8 HTML5 and Living Standards

- HTML5 introduced in ~2008 with a "living standard" approach, abandoning rigid versioning.
- Living standard: constantly updated without version freezes.
- Specification now maintained online at w3.org.
- Practical and responsive to web development pace.
- Allows for more flexibility and quicker browser evolution.

1.9 Generous Syntax Parsing

- Browsers are forgiving: they parse incomplete or incorrect HTML to maximize content rendering.
- This ensures users see as much of the page as possible despite developer errors.
- Contrast to strict languages like C++ which halt on syntax errors.

---

2. XML vs HTML

2.1 XML Overview

- XML = Extensible Markup Language.
- Similar to HTML but stricter syntax.
  - Closing tags are mandatory.

- XHTML is XML-based HTML.

2.2 XML Use Cases

- Used in government and enterprise applications for structured data.
- Useful when correctness and validation are important (e.g., social security data). Primarily used for Data transmission.

2.3 XML vs HTML Syntax

Feature	HTML	XML
Closing Tags	Often optional	Required
Error Tolerance	Very forgiving	Strict
Usage	Web pages (UI)	Structured data, APIs

- XHTML is an XML variant used for well-formed HTML pages.
- Data governance is more reliable in XML (useful in critical fields).

---

3. DOM: Document Object Model

3.1 DOM Overview

- DOM: Tree of HTML content represented as objects.
- JavaScript accesses and manipulates this tree.
- Nodes correspond to elements, text, attributes, etc.

3.2 Key Properties

- Object-Oriented API to interact with the structure of a document.
- Represented in RAM by browsers.

3.3 Operations with DOM

- Navigation: locate elements (`getElementById`, etc.).
- Traversal: visit nodes to search or extract data.
- Modification: change structure, add or remove elements.

3.4 DOM APIs

- Provided by browsers.
- Language-agnostic (usable beyond JavaScript).
- JavaScript is the dominant language using DOM today.

---

4. CSS: Cascading Style Sheets

- Separates content (HTML) from presentation (style).
- Avoids embedding style details directly in HTML.

4.1 Problem with Early HTML Styling

- Direct styling embedded in content:

```
<i>italics</i> <b>bold</b>
```

- Mixed content and presentation, making adaptation difficult.

4.2 Modern Semantic Styling

- Use semantic tags:

```
<em>emphasis</em> <strong>strong emphasis</strong>
```

- Advantages:
  - Makes pages accessible (e.g., screen readers for visually impaired).
  - Adapts better to various devices.

4.3 CSS Inheritance and Cascading

- Styles apply hierarchically.
- Child elements inherit parent's styles unless overridden.

4.4 Sources of Style

Source	Description
Browser	Default styles
User	Custom overrides via browser settings
Author	Defined in the webpage/CSS files

4.5 Style Declaration

- Inline:

```
<span style="font-variant: small-caps;">Styled Text</span>
```

- External:

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

4.6 CSS Syntax Example

```
<span style="font-variant: small-caps;">Sample Text</span>

selector {
  property: value;
}
```

4.5 CSS Usage

- Designed for visual presentation (more design-focused).
- Collaboration between developers (logic) and designers (presentation).

---

5. JavaScript and JSX

5.1 JavaScript Basics

- Dynamic programming language embedded in HTML.
- More dynamic than Python (e.g., runtime decisions).
- Used for interactiveness and logic in webpages.

5.2 Embedding JavaScript in HTML

Two methods:

1. External File:

```
<script src="hello.js"></script>
```

2. Inline Code:

```
<script>
  // code here
</script>
```

5.3 Reasons to Use External Scripts

    - Modularity and code reuse.
    - Easier updates.
    - Support for 3rd-party libraries (no copy-and-paste).
    - Better for copyright compliance.
- Cons:
    - One extra HTTP request
    - More complicated than inline scripts

---

6. JSX: JavaScript XML

6.1 What is JSX?

- A syntax extension to JavaScript used with React.
- Allows writing HTML-like code within JavaScript.
- Transpiled to standard JavaScript behind the scenes.

6.2 JSX Example

```jsx
const header = <h1 lang="en">CS35L Homework</h1>;
```

- JSX expressions can include variables:

```jsx
const course = "CS35L";
const header = <h1>{course} Homework</h1>;
```

6.4 JSX Rendering

```jsx
ReactDOM.render(header, document.getElementById("root"));
```

- Attaches the JSX-generated element to the DOM.

6.5 Nesting and Flexibility

- JSX supports deeply nested elements.
- JavaScript expressions can be embedded within curly braces.

---

7. Client-Server Data Transfer

7.1 JSON: JavaScript Object Notation

- Lightweight data-interchange format.
- Commonly used in web APIs and async data loading.
- Readable and writable directly in JavaScript.
- JSON is to JS as XML is to HTML

JSON (JavaScript Object Notation)

- Trees where labels apply to arcs (key-value pairs).
- Serialized JavaScript object representation.

Example:

```json
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        { "value": "New", "onclick": "CreateNewDoc()" },
        { "value": "Open", "onclick": "OpenDoc()" }
      ]
    }
  }
}
```

7.4 JSON vs XML Tree Representation

Feature	JSON	XML
Label Position	On arcs (keys)	On nodes (element names)
Syntax	Curly braces/arrays	Tags with attributes
Parsing	Native in JavaScript	Requires XML parser

7.5 Comparison to XML

Feature	JSON	XML
Syntax Simplicity	Simpler	Verbose
Readability	Easier horizontally	Harder
Data Orientation	Keys on arcs	Tags on elements (nodes)
Parsability	Built into JavaScript	Requires external library

Format	Description	Example Usage
XML	Verbose, hierarchical	Government, legacy systems
JSON	Lightweight, JavaScript-native	Modern web applications

---

8. Browser Rendering Pipeline

8.1 Stages

1. Parse HTML → create DOM
2. Apply CSS → compute layout
3. Render → display pixels

Optimization Strategies

- Lazy rendering: Start showing content before full parsing completion.
- Skip off-screen elements: Don't render or execute their JS until visible.
- Parallel execution: JavaScript may run while parsing and rendering continues.
- Browsers render partially parsed HTML for responsiveness.

8.3 JavaScript and Rendering

- JavaScript execution may be delayed:
  - If DOM is not visible (e.g., below-the-fold elements).
  - If rendering is incomplete.
- JavaScript and rendering can interleave.
  - Can lead to race conditions if DOM is mutated before render.

Issue	Explanation
Self-modifying DOM	JavaScript modifies the DOM during load
Deferred execution	JS scripts for hidden elements may not run immediately

8.4 Performance Tip

- Be careful where `<script>` tags are placed in HTML.
- Improper placement affects load speed and user experience.

8.5 Testing Concerns

- Fast local networks may hide timing issues.
- Must test on variable latency connections for robustness.

9. Python: Dictionaries and Data Types

9.1 Mapping Types – `dict` (Dictionaries)

- Stores key-value pairs.
- Keys must be hashable and immutable.
- Values can be of any type (mutable/immutable).

9.2 Syntax & Operations

```
d = {"a": 1, "b": 2}
d["c"] = 3            # Add item
print(d["a"])         # Access value
del d["b"]            # Remove item
```

9.3 Core Dictionary Operations

Method	Description
`len(d)`	Number of key-value pairs
`d.clear()`	Remove all pairs
`d.copy()`	Shallow copy of dictionary
`d.items()`	View object of (key, value) tuples
`d.keys()`	View object of keys
`d.values()`	View object of values

9.4 Views and Mutability

- `.items()`, `.keys()`, & `.values()` return view objects:
  - Dynamic: reflect changes in dictionary.
  - Not independently mutable.

9.6 Copying vs Assignment

Assignment:

```
e = d  # e refers to the same dict as d
```

Copy:

```
e = d.copy()  # new dict, shared keys/values
```

9.7 Shallow vs Deep Copy

- Shallow copy:
  - Dict is copied.
  - Mutable values (like lists) are shared.
  - `.copy()` is a shallow copy.
- Deep copy (via `copy.deepcopy()`):
  - Recursively copies everything, including values.

9.8 Dictionaries Are Ordered (Python 3.7+)

- Insertion order is preserved.
- Reassigning a key does not affect its position.
- Deleting and re-adding a key puts it at the end.

9.9 Iterating Over Dictionaries

Example:

```
for key in d:
```

```
for key, value in d.items():
    print(key, value)
```

Summary

This lecture detailed the foundational concepts, history, and technical implementation of HTML, XML, the DOM, CSS, JavaScript, and related technologies. It explored declarative and imperative paradigms of web applications, described HTML syntax rules, different types of elements (void, raw, normal), and discussed the evolution from DTD-based HTML to HTML5's living standard model. The lecture introduced the DOM as an object-oriented representation of HTML/XML trees, explained the role and application of cascading style sheets (CSS) to separate content from presentation, and covered JavaScript and JSX as mechanisms for imperative logic and efficient DOM manipulation. Lastly, it compared data-exchange formats XML and JSON, outlined the browser rendering pipeline for performance optimization, and provided a comprehensive overview of Python dictionaries, focusing on their operations, immutability, view objects, and performance characteristics.