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 for **portability** and **separation of content from form** (presentation).
- HTML inherits:
  - **Content**: What the user wants to convey.
  - **Form**: How content is presented (appearance/style).
- HTML is **declarative**: it specifies what should be displayed, not how to do it.

1.1.1 Semantic HTML

- **Semantic tags** (e.g., `<header>`, `<nav>`, `<main>`, `<article>`, `<footer>`, `<section>`, `<aside>`) describe the meaning of content, not just its appearance.
- **Benefits:**
  - **Accessibility:** Screen readers and assistive tech can better interpret the page.
  - **SEO:** Search engines understand page structure, improving ranking.
  - **Maintainability:** Code is easier to read and update.
- **Example:**

```html
<header>
  <h1>My Blog</h1>
</header>
<main>
  <article>
    <h2>Post Title</h2>
    <p>Content...</p>
  </article>
</main>
<footer>Copyright 2024</footer>
```

1.2 Declarative vs. Imperative

- **Declarative** (HTML): Describes what should appear (structure + content).
- **Imperative** (JavaScript): Describes how things should be done (logic + behavior).
- Original design: Prefer HTML/CSS for content/styling, use JavaScript only for behavior.
- In practice, this separation is often blurred.

1.3 Device Independence and Flexibility

- HTML aims for consistent rendering across devices (laptops, phones, etc.).
- Browsers have flexibility in rendering to improve usability and accessibility.

1.4 HTML Syntax and Elements

1.4.1 Elements and the DOM Tree

- An HTML element = a node in the document's tree structure (DOM).
- Syntax:

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

- Browsers keep the DOM tree in memory; JavaScript manipulates it.
- **ASCII Diagram: DOM Tree Example**

```
<html>
  <body>
    <h1>Hello</h1>
    <p>World</p>
  </body>
</html>
```

```
DOM Tree:
html
 └── body
      ├── h1
      │    └── "Hello"
      └── p
           └── "World"
```

1.4.2 Nesting and Valid Syntax

- Tags must be properly nested:
  - Invalid:

    ```
    <a><b>…</a></b>
    ```

  - Valid:

    ```
    <a><b>…</b></a>
    ```

- Browsers are forgiving: they try to render as much as possible even with errors.

1.4.3 Optional and Void Tags

- Closing tags can sometimes be omitted if unambiguous.
- **Void elements**: Standalone, no children (e.g., `<br>`).
  - `<br>` is equivalent to `<br></br>`, but the short form is preferred.
- Some elements (rare) are **raw text**: can only contain text, not other elements.
- **More Examples:**
  - `<input>`, `<hr>`, `<meta>`, `<link>` are all void elements.
  - `<script>` is a raw text element (can only contain text, not HTML elements).
- **Edge Case:**
  - `<li>` closing tag can be omitted if followed by another `<li>` or if it's the last child.

1.4.4 Attributes

- Elements can have attributes: name-value pairs (strings).
- Syntax:

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

- Attribute names must be unique per element and appear in the opening tag.
- Browsers want to know attributes early for rendering.
- **Boolean attributes:**
  - Some attributes (e.g., `checked`, `disabled`) can appear without a value: `<input checked>` is valid and equivalent to `<input checked="checked">`.

1.4.5 Types of Elements

Type	Characteristics	Example Tags
Normal	Can contain mixed content/children	`<p>`, `<div>`
Raw Text	Can only contain text (rare)	`<script>`
Void	No content or children	` `, `<img>`

1.4.6 Catalogs and DTDs

- Early HTML used a **Document Type Declaration (DTD)** to specify:
  - Allowed elements, their types (void, raw, normal), allowed attributes, valid nesting.
- DTDs defined what trees were valid.
- Used in HTML 1–4, but couldn't keep up with browser innovation.

1.4.7 HTML5 and Living Standards

- HTML5 (~2008) introduced a "living standard" (constantly updated, no version freezes).
- Maintained at w3.org.
- Allows for rapid evolution and flexibility.

1.4.8 Generous Parsing

- Browsers parse incomplete/incorrect HTML to maximize content rendering.
- Contrast: C++ halts on syntax errors; HTML tries to show as much as possible.

---

2. XML vs HTML

2.1 XML Overview

- **XML** = Extensible Markup Language.
- Stricter than HTML: closing tags are mandatory, less forgiving.
- Used for data exchange, especially where correctness/validation is critical (e.g., government, social security).
- **XHTML**: XML-based HTML.
- XML is a generalization of HTML for tree-structured data.
- **Real-world XML use cases:**
  - **SOAP:** Web services protocol.
  - **RSS/Atom:** News feeds.
  - **SVG:** Vector graphics.
  - **Configuration:** `.plist`, `.pom`, `.xml` config files.

2.1.1 XML Tree Example

```
<book>
  <title>CS35L</title>
  <author>Jane Doe</author>
</book>

Tree:
book
 ├── title: "CS35L"
 └── author: "Jane Doe"
```

2.2 XML vs HTML Syntax

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

- XML is used for domain-specific extensions and reliable data governance.

---

3. DOM: Document Object Model

3.1 Overview

- DOM = Tree of HTML (or XML) content as objects in RAM.
- JavaScript (and other languages) can access/manipulate the DOM.
- Nodes: elements, text, attributes, etc.

3.2 DOM Operations

- **Navigation**: Locate elements (e.g., `getElementById`).
- **Traversal**: Visit nodes to search/extract data.
- **Modification**: Change structure, add/remove elements.
- DOM APIs are language-agnostic, but JavaScript is dominant.

3.3 DOM Traversal and Manipulation Example

```
// Find an element by ID and change its text
const el = document.getElementById('greeting');
```

```
el.textContent = 'Hello, world!';

// Add a new element
const p = document.createElement('p');
p.textContent = 'New paragraph.';
document.body.appendChild(p);
```

3.4 DOM Tree Diagram

```
<html>
  └─ body
      ├─ h1
      └─ p
```

---

4. CSS: Cascading Style Sheets

4.1 Motivation

- Separates content (HTML) from presentation (style).
- Early HTML used direct styling (e.g., `<i>`, `<b>`), mixing content and presentation.
- Modern approach: semantic tags (`<em>`, `<strong>`) + CSS for style.
- Accessibility: semantic tags help screen readers, etc.

4.2 CSS Inheritance and Cascading

- Styles apply hierarchically (cascade down the DOM tree).
- Child elements inherit parent styles unless overridden.
- Priority scheme: browser defaults, user overrides, author styles.

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

4.2.1 CSS Specificity and Cascade

- When multiple rules apply, the browser uses **specificity** to decide which wins.
- **Specificity Table:**

Selector Type	Specificity Value
Inline style	1000
ID selector (#id)	100
Class (.class)	10
Element (div, p)	1

- The higher the value, the more specific.
- **Example:**

  ```
  p { color: blue; }          /* specificity: 1 */
  .highlight { color: red; } /* specificity: 10 */
  #main { color: green; }    /* specificity: 100 */
  ```

  If a `<p id="main" class="highlight">` exists, its color will be green.

4.2.2 CSS Rule Override Example

```
<style>
  p { color: blue; }
  .important { color: red; }
</style>
<p class="important">This is red.</p>
```

4.3 CSS Syntax and Usage

- Inline style:

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

- External stylesheet:

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

- CSS syntax:

```css
selector {
  property: value;
}
```

- CSS is more design-focused; collaboration between developers (logic) and designers (presentation).

---

5. JavaScript and JSX

5.1 JavaScript Basics

- Dynamic, imperative language embedded in HTML.
- More dynamic than Python: runtime decisions, harder to predict.
- Used for interactivity and logic in webpages.

5.1.1 JavaScript Event Handling

- JavaScript can respond to user actions (clicks, input, etc.) using event listeners.
- **Example:**

```html
<button id="myBtn">Click me</button>
<script>
  document.getElementById('myBtn').addEventListener('click', function() {
    alert('Button clicked!');
  });
</script>
```

5.1.2 JavaScript and the DOM: Event Flow

- **Event bubbling:** Events propagate from the target element up to the root.
- **Event capturing:** Events can be intercepted on the way down.
- **ASCII Diagram: Event Bubbling**

```
[window]
   |
[document]
   |
[body]
   |
[div]
   |
[button] (event target)
```

  Event bubbles up from button → div → body → document → window.

5.2 Embedding JavaScript

- **External file**:

```
```

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

- **Inline code**:

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

- **Pros of external scripts**:
  - Modularity, code reuse, easier updates, 3rd-party libraries, copyright compliance.
  - Cons: extra HTTP request, more complexity for small scripts.

5.3 DOM Manipulation and JSX

- JavaScript often manipulates the DOM directly (many API calls).
- **JSX**: Shorthand for DOM manipulation, used in React and similar frameworks.
  - JSX is preprocessed into JavaScript function calls.
  - Example:

    ```jsx
    const header = <h1 lang="en">CS35L Homework</h1>;
    ReactDOM.render(header, document.getElementById("root"));
    ```

  - JSX supports embedding JavaScript expressions in curly braces, and deep nesting.
  - You can interpolate variables:

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

- JSX is not required; you can use plain JavaScript DOM APIs, but JSX is more concise.
- **JSX Transformation Example:**

```jsx
// JSX
const el = <h1>Hello</h1>;
// Compiles to
const el = React.createElement('h1', null, 'Hello');
```

---

6. Client-Server Data Transfer: JSON and XML

6.1 JSON: JavaScript Object Notation

- Lightweight, tree-structured data-interchange format.
- Common in web APIs and async data loading.
- Readable/writable in JavaScript; built-in support.
- **JSON is to JS as XML is to HTML**.

Example JSON:

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

6.1.1 Parsing JSON Safely

- **Never use `eval` to parse JSON!**
- Use `JSON.parse()` in JavaScript:

```
const jsonString = '{"a": 1, "b": 2}';
const obj = JSON.parse(jsonString); // {a: 1, b: 2}
```

- **Security risk:** `eval` can execute arbitrary code.

6.1.2 Fetching JSON Example

```
fetch('/api/data')
  .then(response => response.json())
  .then(data => {
    console.log(data);
  });
```

6.1.3 Data Flow Diagram: Server to Client

```
[Server] --(JSON/XML)--> [Network] --(parse)--> [Browser/JS]
```

6.2 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

6.3 Comparison Table

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

---

7. Browser Rendering Pipeline

7.1 Stages

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

7.1.1 Reflow vs. Repaint

- **Reflow:** Changes to layout (e.g., adding/removing elements, changing size) cause the browser to recompute positions and geometry for part or all of the page.
- **Repaint:** Changes only to appearance (e.g., color, visibility) without affecting layout.
- **Reflow is more expensive than repaint.**

Diagram: Reflow vs. Repaint

```
[DOM Change]
   |
   v
[Reflow?] --yes--> [Recompute layout] --+--> [Repaint]
   |                                    |
   no                                   |
   |                                    v
   +--------------------------------> [Repaint only]
```

7.2 Optimization Strategies

- **Lazy rendering**: Start showing content before full parsing.
- **Skip off-screen elements**: Don't render or execute JS for elements not visible yet.
- **Parallel execution**: JavaScript may run while parsing/rendering continues.
- Browsers render partially parsed HTML for responsiveness.
- **Debounce/throttle events:** Limit how often expensive operations run (e.g., on scroll/resize).
- **Minimize layout thrashing:** Batch DOM reads/writes to avoid repeated reflows.

7.3 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 (race conditions possible if DOM is mutated before render).
- **Self-modifying DOM**: JavaScript can modify the DOM during load.
- **Deferred execution**: JS for hidden elements may not run immediately.

7.4 Performance Tips

- Be careful where `<script>` tags are placed; improper placement affects load speed and UX.
- Test on variable latency networks; fast local networks may hide timing issues.

---

8. Python: Dictionaries and Data Types

8.1 Mapping Types – `dict` (Dictionaries)

- Store key-value pairs.
- Keys must be hashable and immutable; values can be any type.
- Dictionaries are glorified hash tables; keys must be immutable for hashing to work.

8.1.1 Hashability and Dict Keys

- **Hashable:** An object is hashable if it has a fixed hash value for its lifetime and can be compared for equality.
- **Valid keys:** Immutable types (str, int, float, tuple of immutables).
- **Invalid keys:** Mutable types (list, dict, set).
- **Example:**

  ```python
  d = {}
  d[(1,2)] = 'ok'    # valid
  d[[1,2]] = 'bad'   # TypeError: unhashable type: 'list'
  ```

8.2 Syntax & Operations

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

8.3 Core Dictionary Operations

Method	Description
`len(d)`	Number of key-value pairs

Method	Description
`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

- `.items()`, `.keys()`, `.values()` return **view objects**:
    - Dynamic: reflect changes in the dictionary.
    - Not independently mutable; just a window into the dict.

8.4 Copying vs Assignment

- **Assignment**: `e = d` (both refer to the same dict; changes via one are seen by the other).
- **Copy**: `e = d.copy()` (new dict, but values are shared if mutable).
- **Shallow copy**: Only the dict is copied; mutable values are shared.
- **Deep copy**: Use `copy.deepcopy()` to recursively copy everything.

8.4.1 Shallow vs Deep Copy Table

Copy Type	What is copied?	Mutable values shared?
Assignment	Reference only	Yes
Shallow copy	Dict only	Yes
Deep copy	Dict and all contents	No

8.5 Dictionary Ordering (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.
- Order matters for reproducibility and debugging.

8.6 Iterating Over Dictionaries

```
for key in d:
    print(key, d[key])

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

- You can convert `.items()`, `.keys()`, `.values()` to lists for random access.

8.7 Performance and Theory

- Dictionary operations are **O(1)** (amortized) due to hashing, as long as memory is finite (word size is fixed).
- If memory could grow without bound, hash table operations would be O(log n) due to word size growth (theoretical note).
- Python dictionaries are not simple hash tables; they maintain insertion order for reproducibility.