HTML and Interactors

Jerry Cain
CS 106AX
November 13, 2023

slide by Eric Roberts

The History of the World Wide Web

- The ideas behind the web are much older than the web itself.
  - In the early 20th century, the Belgian bibliographer Paul Otlet envisioned a universal catalogue that would provide access to all the world's information via an interconnected structure.
  - In 1945, the director of the wartime science program Vannevar Bush published an article entitled "As We May Think," which envisioned an electronic archive of linked information.
  - In the early 1960s, computer visionary Ted Nelson coined the terms *hyperlink* and *hypermedia*.
- The modern web was developed in 1989 by Tim Berners-Lee at CERN, the European particle physics laboratory in Geneva, Switzerland. Berners-Lee developed the first version of the ***Hypertext Markup Language*** (***HTML***).
- Use of the web grew quickly after the release of Mosaic browser in 1993 and Netscape Navigator in 1994.

slide by Eric Roberts

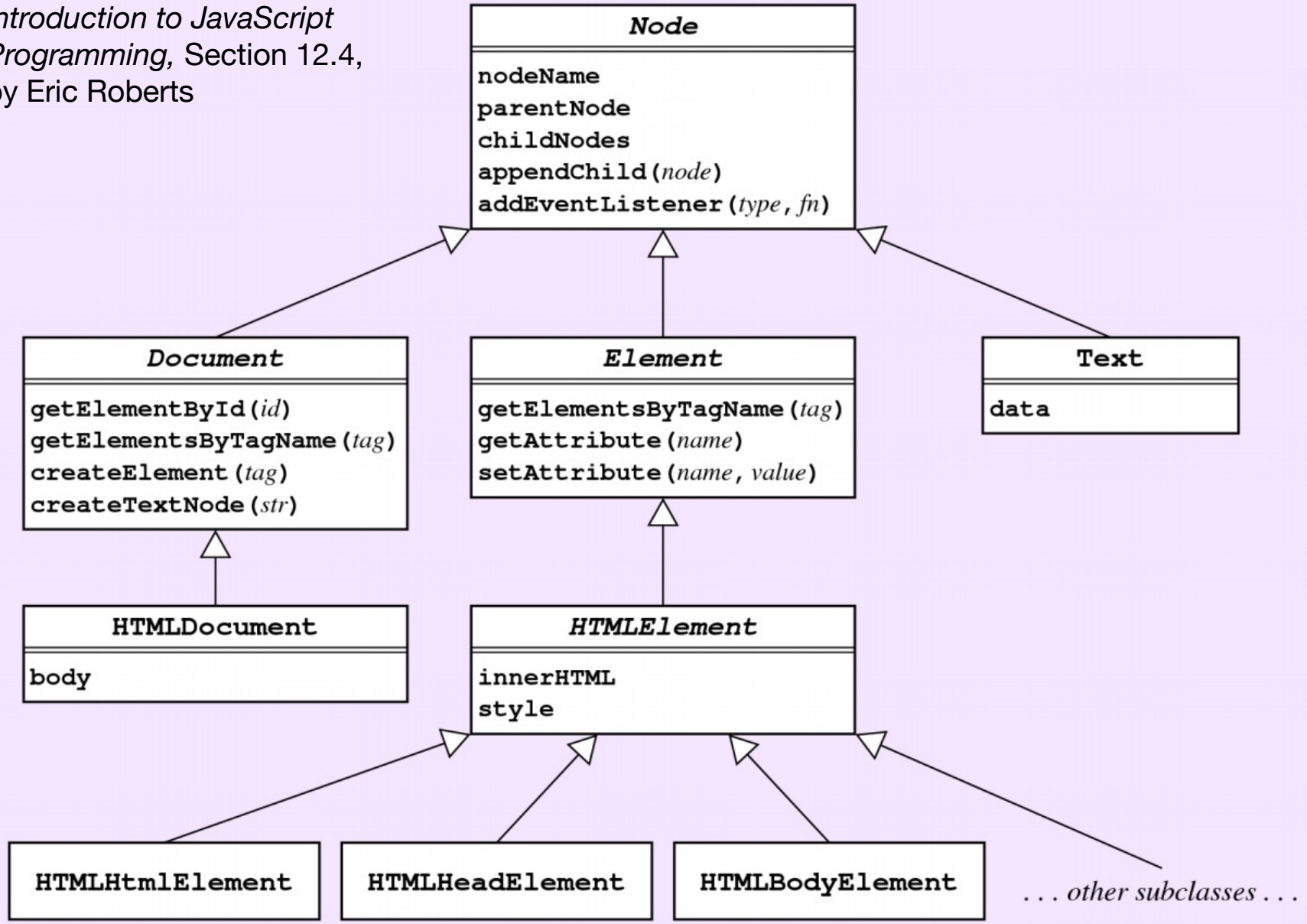
The Document Object Model

- When the browser reads a web page, it first translates the text of the web page into an internal data structure that is easier to manipulate under the control of a JavaScript program. That internal form is called the ***Document Object Model***, or ***DOM***.
- The DOM is structured into a hierarchy of data objects called ***elements***, which usually correspond to a paired set of tags.
- The relationship between the HTML file and the internal representation is like the one between the external data file and the internal data structure in any data-driven program. The browser acts as a driver that translates the HTML into an internal form and then displays the corresponding page.
- The DOM is complicated and requires more than the two weeks of study we have to fully understand it. However, we **can** get a meaningful start given what we know already.

slide by Eric Roberts

Understanding The DOM

Introduction to JavaScript Programming, Section 12.4, by Eric Roberts



Understanding The DOM

- Our first example uses a subset of the DOM that nonetheless achieves something substantial. That subset contains:
  - *Naming an element in the HTML file.* It is often necessary to refer to a specific element in the web page from inside the JavaScript code. By including an **`id`** attribute in the HTML tag, JavaScript code can then find that element by calling **`document.getElementById(`***id***`)`**.
  - *Adding HTML content to an existing element.* We can create new text nodes and HTML elements by calling one of two DOM functions: **`document.createTextNode(`***text***`)`** and **`document.createElement(`***tag***`)`**. We then rely on *element*`.`**`appendChild(`***childnode***`)`**. This last method adds new content under *element* and updates the presentation of the web page.
  - *Attaching event listeners to specific HTML elements*. We install event listeners via *element*`.`**`addEventListener(`***event***`,`** ***callback***`)`**. When *element* detects *event*, it executes **`callback(e)`**.

Example: Do Not Press This Button

File: **do-not-press.html**

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Do Not Press</title>
    <script src="do-not-press.js" type="text/javascript"></script>
  </head>
  <body>
    <button id="button" type="button">Do Not Press</button>
    <div id="digest"></div>
  </body>
</html>
```

- Most of the tags are either boilerplate or irrelevant for the purposes of our discussion here. The two important ones are:
  - **<button>**, which renders a traditional button
  - ** **, which identifies a region where new content can be inserted
- Each of the two contain **id** attributes and are programmatically discoverable via **document.getElementById**.

Example: Do Not Press This Button

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Do Not Press</title>
    <script src="do-not-press.js" type="text/javascript"></script>
  </head>
  <body>
    <button id="button" type="button">Do Not Press</button>
    <div id="digest"></div>
  </body>
</html>
```

```javascript
const WARNING = "Please do not press that button again!";
function onButtonClick(e) {
   let div = document.getElementById("digest");
   let textElement = document.createTextNode(WARNING);
   let breakTag = document.createElement("br");
   div.appendChild(textElement);
   div.appendChild(breakTag);
}

document.addEventListener("DOMContentLoaded", function() {
   let button = document.getElementById("button");
   button.addEventListener("click", onButtonClick);
});
```

Example: Do Not Press This Button

File: **do-not-press.js**

```
const WARNING = "Please do not press that button again!";
function onButtonClick(e) {
   let div = document.getElementById("digest");
   let textElement = document.createTextNode(WARNING);
   let breakTag = document.createElement("br");
   div.appendChild(textElement);
   div.appendChild(breakTag);
}

document.addEventListener("DOMContentLoaded", function() {
   let button = document.getElementById("button");
   button.addEventListener("click", onButtonClick);
}); // DOMContentLoaded event is triggered when DOM is ready
```

- Web programming purists work to minimize the amount of JavaScript appearing in HTML files—that is, they work to separate content and functionality. **`document.addEventListener`** registers a callback to execute once the HTML file has been parsed and the DOM has been constructed. *Note this approach is different than that of your textbook.*
- That callback, when invoked, knows to search for the button element and install a mouse click handler called **`onButtonClick`**.

Example: Interactive To-Do List

File: **to-do-list.html**

```
<html>
  <head>
    <title>To-Do List</title>
    <script src="todo.js" type="text/javascript"></script>
   </head>
   <body>
     Tasks for November 13<sup>th</sup>, 2023:
     <ul id="to-do-list">
       <li>Teach CS106AX and CS107 lectures.</li>
       <li>Take advisees to dinner at Bistro Vida.</li>
     </ul>
     <hr/>
     New item: <input id="item-text" size="60"/>
     <button id="add-item" type="button">Add To List</button>
     <button id="clear-all-items" type="button">Clear</button>
   </body>
</html>
```

- Here's an application designed to track the running list of errands I need to accomplish today. Presumably, I need functionality to add new tasks and delete completed ones.
- To support deletion, we'll need some new DOM methods.

Common DOM Methods and Properties

`document.getElementById(`*id***`)`**	Returns the element with the specified id attribute.
`document.createElement(`*type***`)`**	Creates a new tag element for the tag of the specified type.
`document.createTextNode(`*text***`)`**	Creates a new text element for the provided text string.
element **`.getElementsByTagName(`***name***`)`**	Returns an array of the elements with the specified tag name.
element **`.appendChild(`***node***`)`**	Appends a node to the end of the element's list of children.
element **`.removeChild(`***node***`)`**	Removes a node from the element's list of children.
element **`.parentNode`** *element* **`.lastChild`**	Stores references to a node's parent and last child.

Example: Interactive To-Do List

Here's a list of the four different interactions I'd like to support.

- Add a new item to my to-list, in one of two different ways:
  - Type the new item description and click **`Add To List`**.
  - Type the new item description and hit enter while the input area has the focus.

These require we attach a **`click`** *listener to the button element and a* **`keydown`** *listener to the input element. And because the DOM is updated the same way by each of the two interactions, we want to tap the same code if possible.*

- Remove a list item by double clicking on it.

Doing so requires we attach a **`dblclick`** *event to each* **`li`** *element so that double clicking one removes the item from the DOM. Every element in the DOM stores its parent in a properly named* **`parentNode`***, and a DOM method calls* *element* **`.removeChild(`***node***`)`** *removes the specified node from the DOM.*

- Clear all items by clicking the **`Clear`** button.

This is easiest to implement if we make use of an element's **`lastChild`** *property, which stores a reference to the last child node—the one most efficiently plucked out by a call to element* **`.removeChild(`***node***`)`***. If a node doesn't have any children, then the* **`lastChild`** *property is* **`null`**.

Example: Interactive To-Do List

File: **to-do-list.js (part 1)**

```
function onItemDoubleClick(e) {
   e.target.parentNode.removeChild(e.target);
} // e.target always references source of mouse event

function onAddClick(e) { // e argument is ignored
   let input = document.getElementById("item-text");
   let text = input.value.trim();
   input.value = "";
   if (text.length === 0) return;
   let ul = document.getElementById("to-do-list");
   let li = document.createElement("li");
   li.addEventListener("dblclick", onItemDoubleClick);
   let item = document.createTextNode(text);
   li.appendChild(item);
   ul.appendChild(li);
}

function onClearClick(e) { // e argument is ignored
   let ul = document.getElementById("to-do-list");
   while (ul.childNodes.length > 0) {
      ul.removeChild(ul.lastChild);
   }
}
```

Example: Interactive To-Do List

File: **to-do-list.js (part 2)**

```
const ENTER_KEY = 13;
function onKeyDown(e) {
   if (e.which === undefined) e.which = e.keyCode;
   if (e.which !== ENTER_KEY) return;
   let addButton = document.getElementById("add-item");
   addButton.click(); // programmatically simulate button click
} // event source: e.which or e.keycode, depending on browser

document.addEventListener("DOMContentLoaded", function() {
   let addButton = document.getElementById("add-item");
   addButton.addEventListener("click", onAddClick);

   let clearButton = document.getElementById("clear-all-items");
   clearButton.addEventListener("click", onClearClick);

   let input = document.getElementById("item-text");
   input.addEventListener("keydown", onKeyDown);

   let listItems = document.getElementsByTagName("li");
   for (let i = 0; i < listItems.length; i++) {
      listItems[i].addEventListener("dblclick", onItemDoubleClick);
   }
});
```

Interactive To-Do List Redux

- One of the many drawbacks of the DOM is that the initial specification was vague and incomplete, forcing browser implementations to make decisions that were not portable.
  - Too many web applications have been built now to introduce breaking changes.
  - That's why we need to cope with different event object structures (e.g., mouse events have a **`target`** attribute, key events don't and store the element that triggered the event in either a **`which`** attribute, a **`keyCode`** attribute, or both).
- Our implementation *cannot* blame the DOM for our decision to call **`document.getElementById`** repeatedly to retrieve the same **`button`** and **`ul`** elements. That's on us.
  - Fortunately, we can define callbacks—**`onClearClick`**, **`onKeyUp`**, etc.—as inner functions much as we did for Breakout and Enigma.

Revisited: Interactive To-Do List

File: **to-do-list-improved.js (part 1)**

```
function BootstrapToDoList() {
   /* Variables always visible to inner functions. */
   let ul = document.getElementById("to-do-list");
   let addButton = document.getElementById("add-item");
   let clearButton = document.getElementById("clear-all-items");
   let input = document.getElementById("item-text");

   function onClearClick(e) { // e argument is ignored
      while (ul.childNodes.length > 0) {
         ul.removeChild(ul.lastChild);
      }
   }

   function onItemDoubleClick(e) {
      e.target.parentNode.removeChild(e.target);
   }

   const ENTER_KEY = 13;
   function onKeyDown(e) {
      if (e.which === undefined) e.which = e.keyCode;
      if (e.which !== ENTER_KEY) return;
      addButton.click();
   }
```

Revisited: Interactive To-Do List

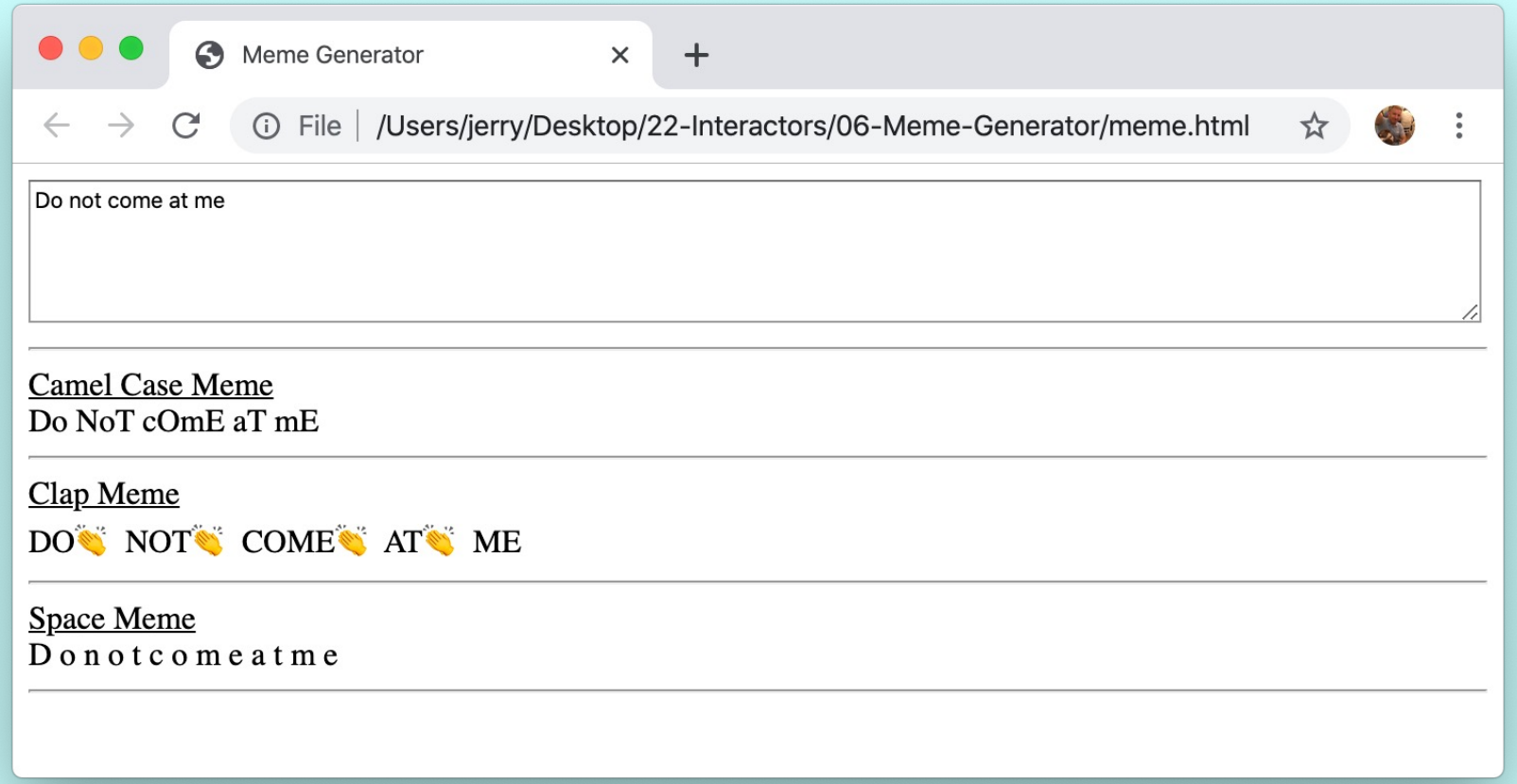
File: **to-do-list-improved.js (part 2)**

```
    function onAddClick(e) { // e argument is ignored
       let text = input.value.trim();
       input.value = "";
       if (text.length === 0) return;
       let li = document.createElement("li");
       li.addEventListener("dblclick", onItemDoubleClick);
       let item = document.createTextNode(text);
       li.appendChild(item);
       ul.appendChild(li);
    }

    addButton.addEventListener("click", onAddClick);
    clearButton.addEventListener("click", onClearClick);
    input.addEventListener("keydown", onKeyDown);
    let listItems = document.getElementsByTagName("li");
    for (let i = 0; i < listItems.length; i++) {
       listItems[i].addEventListener("dblclick", onItemDoubleClick);
    }
}

/* Register BootstrapToDoList to fire when DOM is fully loaded. */
document.addEventListener("DOMContentLoaded", BootstrapToDoList);
```

problem inspiration from Brian Lam

Section Exercise: Meme Generator

Implement a program that creates on-the-fly text memes—specifically, camel case memes, clap memes, and space memes. The memes should update with every **`input`** event fired by the **`textarea`**. The **`textarea`** content is stored in a property called **`value`**.



The End