JavaScript, DOM, and events

Michael Chang
Spring 2023
Plan for today

The DOM
   Traversing, adding, and removing elements

Buttons, inputs and events
   <button>, <label>, <input>, event handlers

Example: unit converter
JS can access the web page using the DOM
Each element is an Element (which is also a Node)
Can walk the tree and add/change/remove elements

Built-in variables
window: info/control the browser window
   The "global object"; you can jam your global vars here
document: access the DOM
    document.head, document.body
Traversing the tree

**.parentElement**
Parent element

**.children**
A *Collection* of children elements

`coll.length, coll[i]`
Access collection as an array

`coll[id] (or coll.id)`
Access elements in collection by id

**Best practice: don't use for generally finding elements**

Will see better way later
But these are good for working with a specific subtree
HTML attributes accessed as JS properties
- `src`, `href`, `id`

`elem.textContent`
- Get/set the text inside an element

**Best practice: avoid `elem.innerHTML`**
- Lets you get/set raw HTML from JS, leads to security issues

**Aside: `alert(message)`**
- Display message in browser

  Recommendation: not great for bigger/production UX, but very useful for debugging/examples/quick things
Adding/removing Elements

`document.createElement(tag)`
Create new element with tag (e.g. "img")

`node.cloneNode(deep)`
Shallow or deep copy of node
Not added to tree

`parent.prepend(child)`
`parent.append(child)`
Add child (element or string to the start/end of parent
Recommendation: don't use `appendChild` and similar Node methods

`.remove()`
Remove node from the tree (still valid object)
**HTML interactors**

**<button>: a button**
Best practice: don't use `<input type="button">`
Children can be anything (text, images)

**<input>: get user input**
Leaf element (no closing tag)
type determines input type (default to text)
  text, checkbox, radio
  Best practice: many useful newer types: number, email, date, ...

**<label>: label an input**
Wrap the `<input>` or use for attribute with an id
Best practice: always use `<label>`; don’t just put text next to the input
**HTML forms**

<form>: wrap a collection of interactors

- Use `<button type="button">`
  - Default is a submit button
- Access forms by id through `document.forms`
- Form instance is a map of interactors (keys are ids)

```javascript
let form = document.forms[formId];
form.myButton.addEventListener("click", (event) => {
  console.log(form.myInput.value);
});
```
Handling events

elem.addEventListener(type, fn)
  type is the event to handle (e.g. click)
  fn is a function to handle the event
  Note: functions can be passed as values!

Event types
  Mouse: click, mouseenter, mouseleave
  Keyboard: keydown, keyup, keypress
  Interaction: change, input, focus, blur

Best practice: semantic elements
  Use the right element, e.g. don't add click handler to paragraph
  Otherwise, may be impossible to use with keyboard/touch/screen reader
const handleClick = (event) => {
  alert("Button was clicked!");
};

let button = document.body.clickme;
button.addEventListener("click", handleClick);
**event argument**

Get info about the event

`event.currentTarget`

The element the listener was added to that triggered the event

Recommendation: `event.target` is slightly different; stick to `currentTarget`

```javascript
const handleClick = (event) => {
  let elem = event.currentTarget;
  elem.textContent = "I was clicked!";
};
```
class App {
    constructor() {
        this._form = document.forms.myForm;
        this._form.myButton.addEventListener("click", this._handler);
    }

    _handler(event) { /* ... */ }
}

(This doesn't work!)
**this keyword**

**Problem**

```javascript
elem.addEventListener(..., this._method);
```

When `_method` is called, this isn’t the instance!

**Cause (summary)**

this gets its value at time of call

- `obj.foo() => this === obj`
- `foo() => this === undefined`
- `let bar = obj.foo; // Not a call, just assigns the fn`  
  `bar(); => this === undefined`
this keyword

Solution

```javascript
elem.addEventListener(...,
    this._method.bind(this));
```
bind sets/"locks" this for future calls

Another solution

In constructor:
```
this._method = this._method.bind(this);
```
Best practice: Do this for all event handlers and callbacks

Not needed for methods called normally
class App {
    constructor() {
        this._handler = this._handler.bind(this);
        this._form = document.forms.myForm;
        this._form.myButton.addEventListener("click",
            this._handler);
    }

    _handler(event) { /* ... */ }
}
Style tips for classes

Bind callbacks in constructor
   To avoid repetition or forgetting

Encapsulation
   Instance variables that "don't make sense" outside of class should be "private"
   But trivial getters/setters are probably unnecessary

Use cases
   "Components": Manage DOM/page functionality
   "Models": Manage data
   Sometimes it makes sense to mix them (if very simple data,
Summary

So far
  Dynamic web pages through DOM manipulation
  User input and event handling

Before next time
  assign1 out, please take a look
    Post on Ed, come to OH with questions

Next week
  More event/DOM examples