JavaScript: DOM & Events

3rd semester @ Erhvervsakademi København

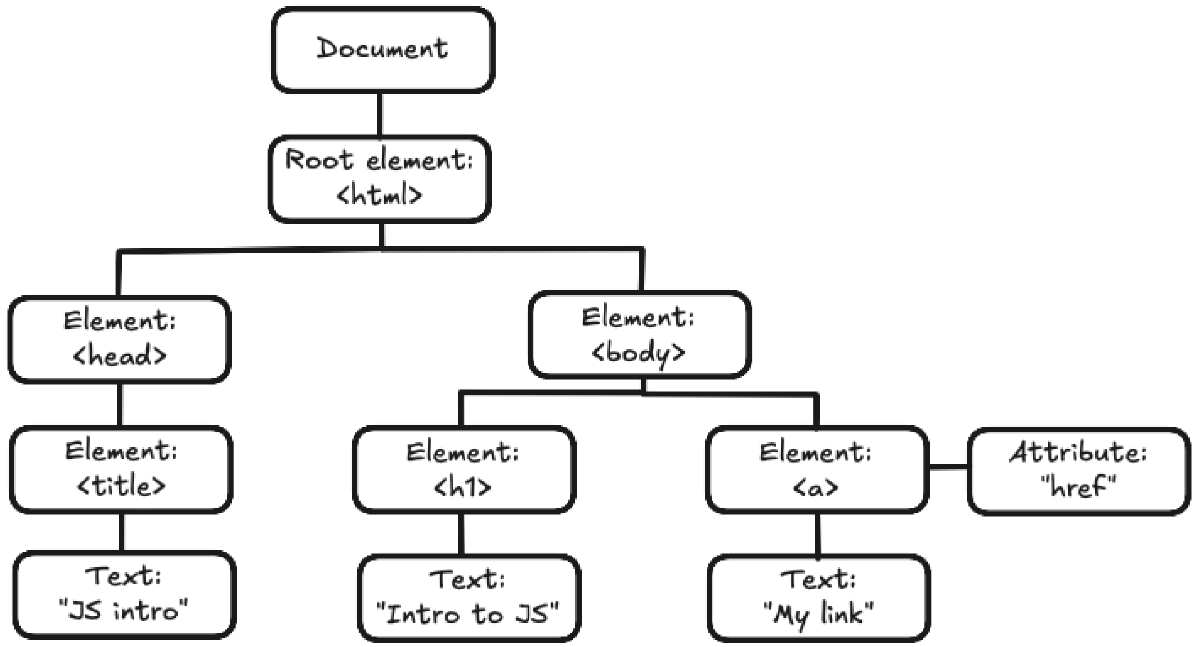
GitHub

Outline:

- DOM & DOM manipulation recap
- Event listeners and event handling
- Forms and form handling with JavaScript
- Event bubbling and capturing
- Event delegation
- Common events

```html
<!DOCTYPE html>
<html>
<head>
    <title>JS intro</title>
</head>
<body>
    <h1>Intro to JS</h1>
    <a href="https://www.example.com">My link</a>
</body>
</html>
```

Document Object Model



Getting DOM elements

> We want to get a reference to a DOM element, so that we can manipulate it or attach event listeners to it.

We can get DOM elements in several ways:

- `const el = document.querySelector("selector")`
- `const els = document.querySelectorAll("selector")`

`selector` is a CSS selector, such as:

- `#id` for an element with a specific id
- `.class` for elements with a specific class
- `tag` for elements with a specific tag name

Manipulating DOM elements

> Once we have a reference to a DOM element, we can manipulate it in various ways.

- **Change content:** `el.textContent = "New text";`
- **Change HTML:** `el.innerHTML = "<span>New HTML</span>";`
- **Change styles:** `el.style.color = "red";`
- **Change attributes:** `el.setAttribute("data-info", "value");`
- **Toggle classes:** `el.classList.toggle("new-class");`
- **Add/remove classes:** `el.classList.add("new-class");` / `el.classList.remove("new-class");`

Adding or removing elements

> We can also add or remove elements from the DOM.

- **Create new element:** `const newEl = document.createElement("div");`
- **Add element to DOM:** `el.appendChild(newEl);`
- **Remove element from DOM:** `newEl.remove();`

Exercises 01: Setup & DOM manipulation

> Part 0: Setup the project and link JavaScript and CSS file.
>
> Part 1: TODO 1 - 6

Event listeners

> We can listen for events on DOM elements, such as clicks:

- **Add event listener:** `el.addEventListener("click", handleClick);`
- **Remove event listener:** `el.removeEventListener("click", handleClick);`

Forms in Thymeleaf

You have previously worked with forms in HTML and Thymeleaf.

```
<form th:action="@{/create}" method="post">
  <input type="text" name="username" />
  <button type="submit">Submit</button>
</form>
```

This form will be submitted to the server, and the page will be reload.

```
POST /create HTTP/1.1
Host: 127.0.0.1:5500
Content-Type: application/x-www-form-urlencoded

username=the_value_entered_in_input
```

Preventing default form submission

> **To handle form submissions with JavaScript, we need to prevent the default behavior of the form** (submitting and reloading the page).

We can do this by calling `event.preventDefault()` in the event handler for the form submission.

But how do we get the `event` object and what is it?

Event object

> When an event occurs, an `event` object is automatically passed to the event handler function. This object contains information about the event, such as the type of event, the target element, and other relevant data.

```javascript
btn.addEventListener("click", handleClick);

function handleClick(event) {
  // We can access event properties here like:
  // event.type, event.target, etc.
  // Prevent default behavior if needed
}
```

Forms with JavaScript

Now we know where the `event` object comes from, we can use it to handle form submissions and access form data.

```html
<form id="todoForm">
  <input type="text" name="todo" placeholder="Enter a todo..."/>
  <button type="submit">Submit</button>
</form>
```

Notice that there is no `method` or `action` attribute on the form. This is because we will handle the form submission with JavaScript.

Forms with JavaScript - Example

```
const todoForm = document.getElementById("todoForm");

todoForm.addEventListener("submit",handleSubmit);

function handleSubmit(event) {
  event.preventDefault();
  const formData = new FormData(event.target);
  const todo = formData.get("todo");
  // ...
}
```

> In this example, we prevent the default form submission, create a `FormData` object from the form, and then get the value of the input field with `formData.get("todo")`.

`FormData` object

> The `FormData` object provides a convenient way to access form data. It allows us to easily get the values of form fields by their `name` attribute.

- We create a `FormData` object by passing the form element to its constructor: `const formData = new FormData(event.target);`
- We can then get the value of a form field using `formData.get("fieldName")`, where `fieldName` is the value of the `name` attribute of the input field.

> **The `name` attribute of the input field must match the string passed to `formData.get()`.**

See the documentation on FormData

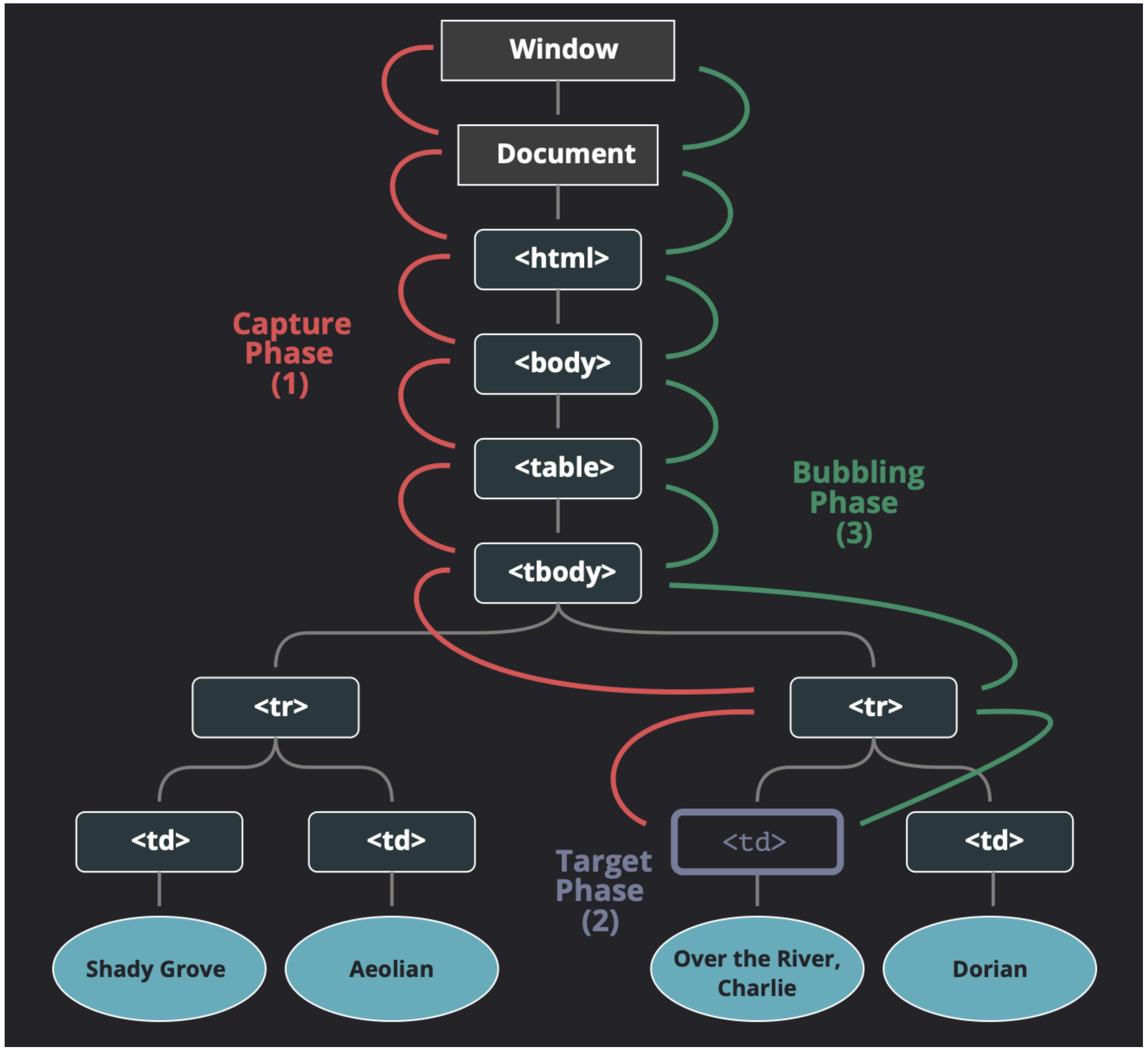
Exercise 01: Form handling and `submit` event

> Part 2: TODO 7 - 13

Event Bubbling and Capturing

> Events propagate through the DOM in two phases: capturing and bubbling.

- **Capturing phase:** The event starts from the root and goes down to the target element.
- **Bubbling phase:** The event bubbles up from the target element back to the root.
- By default, **event listeners listen during the bubbling phase**. We need to opt-in to listen during the **capturing** phase.


Window
Document
<html>
<body>
<table>
<tbody>
<tr>
<tr>
<td>
<td>
<td>
<td>
Shady Grove
Aeolian
Over the River, Charlie
Dorian
Capture Phase (1)
Target Phase (2)
Bubbling Phase (3)

Bubbling and Capturing

> Bubbling and capturing allow us to control the flow of events in the DOM. **Usually, we listen for events during the bubbling phase!**

Benefits of event bubbling:

- It allows us to control the order in which event listeners are executed.
- It **enables event delegation**, where we can listen for events on a parent element and handle events from its children.

Event delegation

Suppose we have a list of items, and we want to handle click events on each item.

```
<ul id="parent">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
  <li>Item 5</li>
  <li>Item 6</li>
  <li>Item 7</li>
</ul>
```

Question: How do we handle events on multiple child elements efficiently?

Event delegation

> **Instead of adding event listeners to each child element, we can add a single event listener to the parent element and use event bubbling to handle events from the children.**

- We add an event listener to the parent element (e.g., the `<ul>`).
- When an event occurs on a child element (e.g., a `<li>`), the event bubbles up to the parent element, where we can handle it.
- We can use `event.target` to determine which child element triggered the event.

Event delegation - Example

```
const parent = document.querySelector("#parent");

parent.addEventListener("click", handleClick);

function handleClick(event) {
  const clickedItem = event.target; // This will be the <li> that was clicked
  console.log("Clicked item:", clickedItem.textContent);
}
```

Clicking on any of the `<li>` items will trigger the event listener on the parent `<ul>`, and we can access the clicked item through `event.target`.:

```
Clicked item: Item 1
```

Event delegation - `.closest()`

> Sometimes we want to find the nearest ancestor of the clicked element that matches a specific selector. We can use the `closest()` method for this.

Suppose we have a more complex structure inside our list items:

```html
<ul id="parent">
    <li data-id="101">
        <span style="color: green">Item 1:</span>
        <span style="color: blue;">Some description</span>
    </li>
    <li data-id="102">
        <span style="color: green">Item 2:</span>
        <span style="color: blue;">Some description</span>
    </li>
  <!--- more items -->
</ul>
```

Event delegation - `closest()` example

```
document.querySelector("#parent").addEventListener("click", handleClick);

function handleClick(event) {
    const li = event.target.closest("li");
    console.log("Clicked item:", li.getAttribute("data-id"));
}
```

Even if we click on the inner `<span>` elements, we can still find the closest `<li>` ancestor and access its `data-id` attribute.

```
Clicked item: 101
```

Events

There exist many types of events, that we can listen for and attach event listeners to.

- Mouse events: `click`, `dblclick`, `mouseover`, `mouseout`, `mousemove`, `mousedown`, `mouseup`
- Keyboard events: `keydown`, `keyup`, `keypress`
- Form events: `submit`, `change`, `input`, `focus`, `blur`
- Window events: `load`, `resize`, `scroll`, `unload`, `contextmenu`
- Clipboard events: `copy`, `cut`, `paste`

> You should at least know **`click`** and **`submit`** events. The rest are not mandatory, but you can experiment with them if you want to.

Exercise 01: Event delegation

> Part 3: TODO 14 - 21

Exercise 02: Guess My Number Game

> Implement a game where the user has to guess a random number between 1 and 100. After each guess, the user receives feedback on whether their guess is too low, too high, or correct. The game keeps track of the number of attempts, and the user can restart the game after guessing correctly.