DOM MANIPULATION & EVENTS

A. Document Object Model (DOM)

The DOM (or Document Object Model) is a tree-like representation of the contents of a webpage. Example:

```html
<div id="container">
  <div class="display"></div>
  <div class="controls"></div>
</div>
```

In the above example, the `<div class="display"></div>` is a "child" of `<div id="container"></div>` and a "sibling" to `<div class="controls"></div>`. Think of it like a family tree. `<div id="container"></div>` is a **parent**, with its **children** on the next level, each on their own "branch".

1. Targeting node with selectors
   You can use a combination of CSS-style selectors(i.e., .display, #container) and relationship properties(i.e., firstElementChild or lastElementChild) to target the nodes you want.

```js
// selects the #container div (don't worry about the syntax,
const container = document.querySelector("#container");

// selects the first child of #container => .display
console.dir(container.firstElementChild);
```

B. DOM Methods (exericse at odin web)

1. Query Selectors
   - `element.querySelector(selector)` - returns a reference to the first match of *selector*.
   - `element.querySelectorAll(selectors)` - returns a "NodeList" containing references to all of the matches of the *selectors*.

   There are several other, but we won't be going over them now.
   Note:

   It's important to remember that when using querySelectorAll, the return value is **not** an array. It looks like an array, and it somewhat acts like an array, but it's really a "NodeList". The big distinction is that several array methods are missing from NodeLists. One solution, if problems arise, is to convert the NodeList into an array. You can do this with `Array.from()` or the spread operator.

   Spread operator link: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

Note: There are older methods available for grabbing element references, such as Document.getElementById() & Document.getElementsByTagName(). These two work better in older browsers than the modern methods like querySelector(), but are not as convenient.

2. Element Creation
   - `document.createElement(tagName, [options])` - creates a new element of tag type tagName.

   [options] in this case means you can add some optional parameters to the function, you don't worry about these at this point.

```
1 | const div = document.createElement("div");
```

This function does NOT put your new element into the DOM - it creates it in memory. This is so that you can manipulate the element (by adding styles, classes, ids, text, etc.) before placing it on the page. You can place the element into the DOM with one of the following methods.

3. Append Elements
   - `parentNode.appendChild(childNode)` - appends *childNode* as the last child of *parentNode*.
   - `parentNode.insertBefore(newNode, referenceNode)` - inserts *newNode* into *parentNode* before *referenceNode*.

4. Remove Elements
   - `parentNode.removeChild(child)` - removes *child* from *parentNode* on the DOM and returns a reference to *child*.

5. Altering Elements

   When you have a reference to an element, you can use that reference to alter the element's own properties. This allows you to do many useful alterations, like adding, removing, or altering attributes, changing classes, adding inline style information, and more.

```
// creates a new div referenced in the variable 'div'
const div = document.createElement("div");
```

6. Adding inline style

```
// adds the indicated style rule to the element in the div varial
div.style.color = "blue";

// adds several style rules
div.style.cssText = "color: blue; background: white;";

// adds several style rules
div.setAttribute("style", "color: blue; background: white;");
```

When accessing a kebab-cased CSS property like background-color with JS, you will need to either use camelCase with dot notation or bracket notation. When using bracket

notation, you can use either camelCase or kebab-case, but the property name must be a string.

```
// dot notation with kebab case: doesn't work as it attempts to
// equivalent to: div.style.background - color
div.style.background-color;

// dot notation with camelCase: works, accesses the div's backg
div.style.backgroundColor;

// bracket notation with kebab-case: also works
div.style["background-color"];

// bracket notation with camelCase: also works
div.style["backgroundColor"];
```

There is another common way to dynamically manipulate styles on your document:

```
<style>
  .highlight {
    color: white;
    background-color: black;
    padding: 10px;
    width: 250px;
    text-align: center;
  }
</style>
para.setAttribute("class", "highlight");
```

The first method(element.style) takes less setup and is good for simple uses, whereas the second method(.setAttribute) is as you start building larger and more involved apps, you will probably start using the second method more, but it is really up to you.

7. Editing Attributes
   - *element*.setAttribute() : sets a new value to an attribute. If the attribute does not exist, it create the attribute with the value. Example:

     ```
     element.setAttribute("class", "red-text");
     ```

- *element*.getAttribute() : returns the value of an element's attribute. Example:
  `let text = element.getAttribute("class");`
  Get the value of the class attribute of an element.
- *element*.removeAttibute() : removes an attribute from an element. Example:
  `div.removeAttribute("id");` remove id attribute from div.

HTML attribute(full): https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes

8. Working with classes

```
1  // adds class "new" to your new div
2  div.classList.add("new");
3
4  // removes "new" class from div
5  div.classList.remove("new");
6
7  // if div doesn't have class "active" then add it, or if it does, then remove it
8  div.classList.toggle("active");
```

It is often standard (and cleaner) to .toggle a CSS style rather than adding and removing inline CSS.

9. Adding text content
   `div.textContent = "Hello World!";`
   creates a text node containing "Hello World!" and inserts it in div.

10. Adding HTML content
    `div.innerHTML = "<span>Hello World!</span>";`
    renders the HTML inside div

Note: using textContent is preferred over innerHTML for adding text, as innerHTML should be used sparingly to avoid potential security risks. To understand the dangers of using innerHTML, watch this:
https://youtu.be/ns1LX6mEvyM?si=wHV_xaY6MUupR6Ha

Keep in mind that the JavaScript does not alter your HTML, but the DOM - your HTML file will look the same, but the JavaScript changes what the browser renders.

Your JavaScript, for the most part, is run whenever the JS file is run or when the script tag is encountered in the HTML. If you are including your JavaScript at the top of your file, many of these DOM manipulation methods will not work because the JS code is being run *before* the nodes are created in the DOM. The simplest way to fix this is to include your JavaScript at the bottom of your HTML file so that it gets run after the DOM nodes are parsed and created.

Alternatively, you can link the JavaScript file in the `<head>` of your HTML document. Use the `<script>` tag with the `src` attribute containing the path to the JS file, and include the `defer` keyword to load the file *after* the HTML is parsed, as such:

```
<head>
  <script src="js-file.js" defer></script>
</head>
```

C. Events

Events are actions that occur on your webpage, such as mouse-clicks or key-presses. Using JavaScript, we can make our webpage listen to and react to these events. There are three method:

- You can specify function attributes directly on your HTML elements.
- You can set properties in the form of `on<eventType>`, such as `onclick` or `onmousedown`, on the DOM nodes in your JavaScript.
- You can attach event listeners to the DOM nodes in your JavaScript.

Event listeners are definitely the preferred method, but you will regularly see the others in use:

- Method 1

```
<button onclick="alert('Hello World')">Click Me</button>
```

We can only set one “onclick” property per DOM element, so we’re unable to run multiple separate functions in response to a click event using this method. (This solution is less than ideal because we’re cluttering our HTML with JavaScript).

- Method 2

```html
<!-- the HTML file -->
<button id="btn">Click Me</button>
```

```js
// the JavaScript file
const btn = document.querySelector("#btn");
btn.onclick = () => alert("Hello World");
```

We still have the problem that a DOM element can only have one “onclick” property. (This is a little better. We’ve moved the JS out of the HTML and into a JS file).

- Method 3

```html
<!-- the HTML file -->
<button id="btn">Click Me Too</button>
```

```js
// the JavaScript file
const btn = document.querySelector("#btn");
btn.addEventListener("click", () => {
  alert("Hello World");
});
```

Now we allow multiple event listeners if the need arises.

- Note that all three of these methods can be used with named functions like so:

```html
<!-- the HTML file -->
<!-- METHOD 1 -->
<button onclick="alertFunction()">CLICK ME BABY</button>
```

```
// the JavaScript file
// METHOD 1
function alertFunction() {
  alert("YAY! YOU DID IT!");
}
```

```
<!-- the HTML file -->
<!-- METHODS 2 & 3 -->
<button id="btn">CLICK ME BABY</button>
```

```
// the JavaScript file
// METHODS 2 & 3
function alertFunction() {
  alert("YAY! YOU DID IT!");
}
const btn = document.querySelector("#btn");

// METHOD 2
btn.onclick = alertFunction;

// METHOD 3
btn.addEventListener("click", alertFunction);
```

Using named functions can clean up your code considerably, and is a really good idea if the function is something that you are going to want to do in multiple places.

```
<button id="btn">CLICK ME BABY</button>
```

```
btn.addEventListener("click", function (e) {
  e.target.style.background = "blue";
});
```

The e parameter in that callback function(function as argument) contains an object that references the event itself.

- Attaching listeners to Groups of nodes

```
<div id="container">
  <button id="1">Click Me</button>
  <button id="2">Click Me</button>
  <button id="3">Click Me</button>
</div>
```

Lanjutannya di bawah(di page selanjutnya)

```
// buttons is a node list. It looks and acts much like an array.
const buttons = document.querySelectorAll("button");

// we use the .forEach method to iterate through each button
buttons.forEach((button) => {
  // and for each one we add a 'click' listener
  button.addEventListener("click", () => {
    alert(button.id);
  });
});
```

- Some useful events include
  - click
  - change
  - dblclick
  - keydown
  - keyup

  Full: https://www.w3schools.com/jsref/dom_obj_event.asp

D. Event Flow (summary is below E. Event Object)

src: https://www.javascripttutorial.net/javascript-dom/javascript-events/

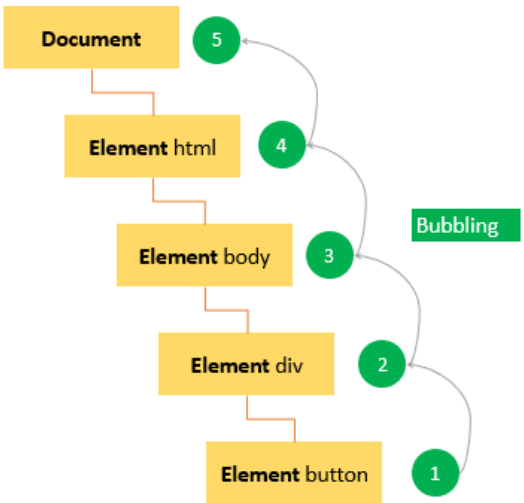
Assuming that you have the following HTML document:

```
<!DOCTYPE html>
<html>
<head>
    <title>JS Event Demo</title>
</head>
<body>
    <div id="container">
        <button id='btn'>Click Me!</button>
    </div>
</body>
```

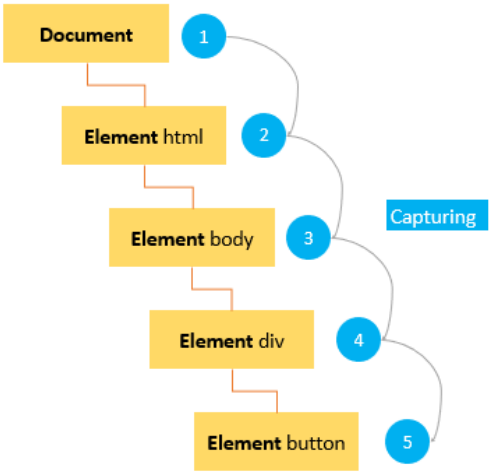
There are two main event models:

1. Event bubbling

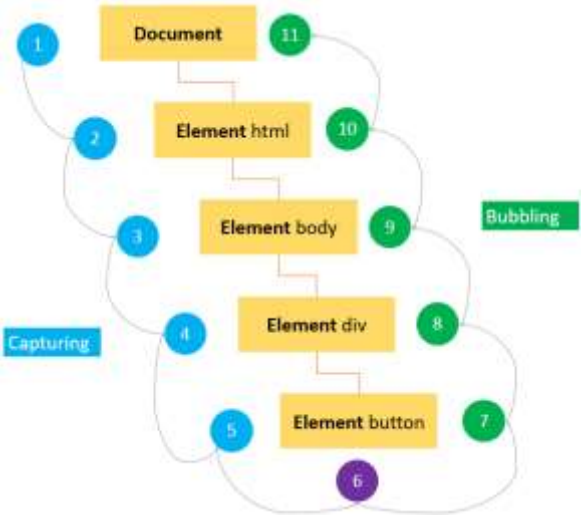
   Event starts at the most specific element and then flows upward toward the least specific element (the document or even window in modern browsers). When you click the button, the 'click' event occurs in the following order:



2. Event capturing
   Event starts at the least specific element and flows downward toward the most specific element. When you click the button, the ‘click’ event occurs in the following order:



3. DOM level 2 event flow
   First, event capturing occurs. Then, the actual target receives the event. Finally, event bubbling occurs.



E. Event Object

When the event occurs, the web browser passes an ‘Event’ object to the event handler:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click', function(event) {
    console.log(event.type);
});
```

Output:

```
'click'
```

Event handler is a function that is called when an event occurs.

Event object adalah sebuah objek yang dikirimkan ke event handler ketika sebuah event terjadi. Objek ini berisi informasi tentang event tersebut, seperti tipe event, elemen yang memicu event, dan berbagai properti lainnya yang berkaitan dengan event tersebut.

Objek event dalam JavaScript memiliki banyak properti dan metode yang berguna, seperti:

- `type`: Tipe dari event (misalnya, 'click', 'mouseover', dll.).
- `target`: Elemen yang memicu event.
- `preventDefault()`: Mencegah tindakan default yang biasanya dilakukan oleh browser saat event terjadi.
- `stopPropagation()`: Menghentikan propagasi event ke elemen-elemen lain.

Most commonly used properties and methods of the 'event' object:

Property / Method	Description
bubbles	true if the event bubbles
cancelable	true if the default behavior of the event can be canceled
currentTarget	the current element on which the event is firing
defaultPrevented	return true if the preventDefault() has been called.
detail	more information about the event
eventPhase	1 for capturing phase, 2 for target, 3 for bubbling
preventDefault()	cancel the default behavior for the event. This method is only effective if the `cancelable` property is true
stopPropagation()	cancel any further event capturing or bubbling. This method only can be used if the `bubbles` property is true.
target	the target element of the event
type	the type of event that was fired

Note: 'event' object is only accessible inside the event handler.

- preventDefault() : To prevent the default behavior of an event.
  For example, when you click a link, the browser navigates you to the URL specified in the href attribute:

```
<a href="https://www.javascripttutorial.net/">JS Tutorial</a>
```

  However, you can prevent this behavior by using the 'preventDefault()' method of the 'event' object:

```
let link = document.querySelector('a');

link.addEventListener('click',function(event) {
    console.log('clicked');
    event.preventDefault();
});
```

- stopPropagation() : stops the flow of an event through the DOM tree. However, it does not stop the browser's default behavior. Example:
  Without the 'stopPropagation()' method, you would see two messages on the Console window.

```
let btn = document.querySelector('#btn');

btn.addEventListener('click', function(event) {
    console.log('The button was clicked!');
    event.stopPropagation();
});

document.body.addEventListener('click',function(event) {
    console.log('The body was clicked!');
});
```

  However, the 'click' event never reaches the 'body' because the 'stopPropagation()' was called on the 'click' event handler of the button.
  Summary D & E:

- An event is an action occurred in the web browser e.g., a mouse click.
- Event flow has two main models: event bubbling and event capturing.
- DOM Level 2 Event specifies that the event flow has three phases: event bubbling, the event occurring at the exact element, and event capturing.
- Use `addEventListener()` to register an event that connects an event to an event listener
- The `event` object is accessible only within the event listener.
- Use `preventDefault()` method to prevent the default behavior of an event, but does not stop the event flow.
- Use `stopPropagation()` method to stop the flow of an event through the DOM tree, but does not cancel the browser default behavior.

F. Page Load Events

- When you open a page, the following events occur in sequence

- `DOMContentLoaded` – the browser fully loaded HTML and completed building the DOM tree. However, it hasn't loaded external resources like stylesheets and images. In this event, you can start selecting DOM nodes or initialize the interface.
- `load` – the browser fully loaded the HTML and external resources like images and stylesheets.

- When you leave the page, the following events fire(occur) in sequence:

- `beforeunload` – fires before the page and resources are unloaded. You can use this event to show a confirmation dialog to confirm if you want to leave the page. By doing this, you can prevent data loss in case the user is filling out a form and accidentally clicks a link that navigates to another page.

- `unload` – fires when the page has completely unloaded. You can use this event to send the analytic data or to clean up resources.

The following example illustrates how to handle the page load events:

```
<body>
    <script>
        addEventListener('DOMContentLoaded', (event) => {
            console.log('The DOM is fully loaded.');
        });

        addEventListener('load', (event) => {
            console.log('The page is fully loaded.');
        });

        addEventListener('beforeunload', (event) => {
            // show the confirmation dialog
            event.preventDefault();
            // Google Chrome requires returnValue to be set.
            event.returnValue = '';
        });

        addEventListener('unload', (event) => {
            // send analytic data
        });
    </script>
</body>
```

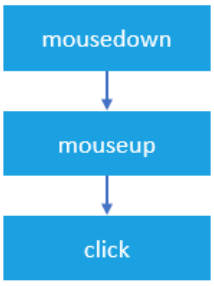
G. Mouse Events (https://www.javascripttutorial.net/javascript-dom/javascript-mouse-events/)

DOM Level 3 events define nine mouse events.

- mousedown, mouseup, and click

  When you click an element, there are no less than three mouse events fire in the following sequence:

1. The `mousedown` fires when you press the mouse button on the element.
2. The `mouseup` fires when you release the mouse button on the element.
3. The `click` fires when one `mousedown` and one `mouseup` detected on the element.



- dblclick

  The ‘dblclick’ event fires when you double-click over an element. In practice, you rarely use the ‘dblclick’ event. The ‘dblclick’ event has four distinct events fired in the following order:

1. `mousedown`
2. `mouseup`
3. `click`

4. `mousedown`
5. `mouseup`
6. `click`
7. `dblclick`

- mousemove

  The ‘mousemove’ event fires repeatedly whenever you move the mouse cursor around an element. This ‘mousemove’ event fires many times per second as the mouse is moved around, even if it is just by one pixel. (This may lead to a performance issue if the event handler function is complex).

  To avoid the performance issue, it is a good practice to add ‘mousemove’ event handler only when you need it and remove it as soon as it is no longer needed, like this:

```
element.onmousemove = mouseMoveEventHandler;
// ...
//  later, no longer use
element.onmousemove = null;
```

- mouseover/mouseout

The `mouseover` fires when the mouse cursor is outside of the element and then moves inside the boundaries of the element.

The `mouseout` fires when the mouse cursor is over an element and then moves another element.

- mouseenter/mouseleave

The `mouseenter` fires when the mouse cursor is outside of an element and then moves inside the boundaries of the element.

The `mouseleave` fires when the mouse cursor is over an element and then moves to the outside of the element’s boundaries.

Both `mouseenter` and `mouseleave` does not bubble and does not fire when the mouse cursor moves over descendant elements.

Registeing mouse event handlers:

```
<button id="btn">Click Me!</button>
```

- Recommended

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',(event) => {
    console.log('clicked');
});
```

- Okay

```
let btn = document.querySelector('#btn');

btn.onclick = (event) => {
    console.log('clicked');
};
```

- In old system, you may find

```
<button id="btn" onclick="console.log('clicked')">Click Me!</button>
```

Detecting mouse buttons

The 'event' object passed to the mouse event handler has a property called 'button' that indicates which mouse button was pressed on the mouse to trigger the event.

The mouse button is represented by a number:

- 0: the main mouse button is pressed, usually the left button.
- 1: the auxiliary button is pressed, usually the middle button or the wheel button.
- 2: the secondary button is pressed, usually the right button.
- 3: the fourth button is pressed, usually the Browser Back button.
- 4: the fifth button is pressed, usually the *Browser Forward* button.

Example & Demo:

Click me with any mouse button: left, right, middle, ...

Right mouse button clicked.

```
<button id="btn">Click me with any mouse button: left, right, middle, ...</buttor
<p id="message"></p>
<script>
    let btn = document.querySelector('#btn');

    // disable context menu when right-mouse clicked
    btn.addEventListener('contextmenu', (e) => {
        e.preventDefault();
    });

    // show the mouse event message
    btn.addEventListener('mouseup', (e) => {
        let msg = document.querySelector('#message');
        switch (e.button) {
            case 0:
                msg.textContent = 'Left mouse button clicked.';
                break;
            case 1:
                msg.textContent = 'Middle mouse button clicked.';
                break;
            case 2:
                msg.textContent = 'Right mouse button clicked.';
                break;
            default:
                msg.textContent = `Unknown mouse button code: ${event.button}`;
        }
    });
</script>
```

Modifier keyboard events

When you click an element, you may press one or more modifier keys: Shift, Ctrl, Alt, and Meta(windows key/command key on apple keyboard).

To detect if these modifier keys have been pressed, you can use the event object(‘true’ if the key is being held down or ‘false’ if the key is not pressed.) passed to the mouse event handler.

```
<button id="btnKeys">Click me with Alt, Shift, Ctrl pressed</button>
<p id="messageKeys"></p>

<script>
    let btnKeys = document.querySelector('#btnKeys');

    btnKeys.addEventListener('click', (e) => {
        let keys = [];

        if (e.shiftKey) keys.push('shift');
        if (e.ctrlKey) keys.push('ctrl');
        if (e.altKey) keys.push('alt');
        if (e.metaKey) keys.push('meta');

        let msg = document.querySelector('#messageKeys');
        msg.textContent = `Keys: ${keys.join('+')}`;
    });
</script>
```

Click me with Alt, Shift, Ctrl pressed

Keys: shift

Screen Coordinates Events

- screenX dan screenY : memberikan koordinat X dan Y posisi relatifterhadap seluruh layar perangkat.
- clientX dan clientY : memberikan koordinat X dan Y posisi mouse relatif terhadap jendela aplikasi atau elemen HTML di mana event tersebut terjadi.

```
<p>Move your mouse to see its location.</p>
<div id="track"></div>
<p id="log"></p>

<script>
    let track = document.querySelector('#track');
    track.addEventListener('mousemove', (e) => {
        let log = document.querySelector('#log');
        log.innerText = `
        Screen X/Y: (${e.screenX}, ${e.screenY})
        Client X/Y: (${e.clientX}, ${e.clientY})`
    });
</script>
```

Move your mouse to see its location.

Screen X/Y: (52, 407)
Client X/Y: (7, 158)

Summary

- DOM Level 3 defines nine mouse events.
- Use `addEventListener()` method to register a mouse event handler.
- The `event.button` indicates which mouse button was pressed to trigger the mouse event.
- The modifier keys: alt, shift, ctrl, and meta (Mac) can be obtained via properties of the event object passed to the mouse event handler.

- The `screenX` and `screenY` properties return the horizontal and vertical coordinates of the mouse pointer in screen coordinates.
- The `clientX` and `clientY` properties of the `event` object returns horizontal and vertical coordinates within the application's client area at which the mouse event occurred.

H. **Keyboard Events** (https://www.javascripttutorial.net/javascript-dom/javascript-keyboard-events/)
I. **Event Delegation** (https://www.javascripttutorial.net/javascript-dom/javascript-event-delegation/) (bisa juga seperti page 8 bagian atas di file ini)
J. **dispatchEvent** (https://www.javascripttutorial.net/javascript-dom/javascript-dispatchevent/) dispatchEvent() digunakan untuk mensimulasikan klik pada button secara programatik (dalam kode), seolah-olah pengguna mengklik button tersebut.
K. **Custom Events** (https://www.geeksforgeeks.org/javascript-custom-events/) Just supplement(not mandatory): https://www.javascripttutorial.net/javascript-dom/javascript-custom-events/