Lab 02

In this lab we explore JavaScript (JS), as well as a bit more advanced HTML / CSS features.

Project's Current Directoyry Structure

```
lab_02
|___index.html
|___css
|   |___style.css
|___js
|   |___main.js
|___html
|___assets
    |___fonts
        |___
```

Overview

- Events and Event Listeners
  - Using JS to access and modify the DOM tree.
  - Using JS to load page components / parts.
  - Representing DOM elements JS-wise.
- More Advanced Styling
  - Load fonts and other assets from CSS.
  - Apply CSS rules conditionally based on DOM element content.
- A Deeper Dive into JS
  - Adding buttons and the `onclick` / `click` event.
  - Predevnting default behaviours.
  - Measuring time with JS.
- Flexing our HTML Muscle
  - Use flex-boxes to implement responsive page-structure.
  - Create a simple navigation bar and footer.

Saving Up Some HTML Code

In order to generate our minesweeper 8x8 grid, we have repeated 64 times the same line. While this works, it is not quite efficient, as with any code repetition. So, how about rendering all those identical cells with a more dynamic way? To do so, we will need the following JS code snippet, stored in file `./js/main.js`.

```
function loadGrid(size = 8) {
    const gridContainer = document.getElementById("grid-container");
    let cell;
    for (let i = 0; i < size; i++) {
        for (let j = 0; j < size; j++) {
```

```
            cell = document.createElement("div");
            cell.id = i + "-" + j; // Just a dummy way to identify each cell.
            cell.classList.add("minesweeper-cell");
            gridContainer.appendChild(cell);
        }
    }
}

function windowOnLoad() {
    loadGrid();
}

window.addEventListener("load", windowOnLoad);
```

Before you proceed, try for a moment to understand what is going on above.

[Take your time…]

Now, let's focus on `loadGrid()` at first. In JS, we declare functions simple by using the keyword `function`, much like in Python we do so by using `def`. Also, as in Python, JS functions need not return a certain type of objects; i.e., they may return anything, if at all. The same applies to function arguments, whose type we need not declare (in general, there is no type decalration in JS). So, this line:

```
function loadGrid(size = 8) { ... }
```

declares a function named `loadGrid` which accepts a single argument, `size`, which has a default value set to `8`. Then, we proceed to define this functions behaviour, by first fetching a DOM element:

```
const gridContainer = document.getElementById("grid-container");
```

By this line, we declare a *constant*, named `gridContainer`, which is actually a reference to a DOM element (i.e., some element residing in our HTML code in some way) which has id `grid-container`. If no such element exists, the (built-in) function `document.getElementById()` returns `undefined` (well-behaving JS equivalent to `null`, even if JS has its own `null` as well). Thus, we have established a connection between the DOM tree of our webpage and our JS script.

With the line below we just declare a variable named `cell` for future use:

```
let cell;
```

In general, there are three ways to declare stuff in JS:

- `const`, which refers to contant entities, i.e., those whose value does not change with time.
- `let`, which refers to variable entities **within the current scope**.
- `var`, which refers to variable entities **within the global scope**.

As a rule of thumb, use `const` whenever possible and `let` in any other cases (you will barely need `var` most of the times).

Now it's time to make a deep dive into the core of `loadGrid()`:

```
for (let i = 0; i < size; i++) {
    for (let j = 0; j < size; j++) {
        cell = document.createElement("div");
        cell.id = i + "-" + j; // Just a dummy way to identify each cell.
        cell.classList.add("minesweeper-cell");
        gridContainer.appendChild(cell);
    }
}
```

In this nested for-loop (observe how JS allows for a C/C++ loop syntax), we:

- create a `div` element, by using the built-in `document.createElement()` function;
- assign it an id (this will be needed in case we need to reference a certain cell in the future);
- add `minesweeper-cell` CSS-class to its classlist (so that this `div` has the desired style);
- append our newly born cell to its container, i.e., `gridContainer`.

Now, just defining `loadGrid()`, we are ready to use it wherever we need to. To do so, we have to attach our function to a **DOM event**. You can think of events as… events that take place during the lifetime of a web page / app. So, before attaching `loadGrid()` to some event, we have to think when we want our grid to be loaded first.

[Take some time to think…]

Evidently, we want our grid to show up whenever our page has loaded. Luck is on our side today, since JS offers exactly this event, coming under the name `load`. To attach `loadGrid()` to this event, we can simply use this line:

```
window.addEventListener("load", loadGrid);
```

Observe how `window.addEventListener()` accepts the event's name as a string as its first argument and the **name** of the function to be called on that event as a second argument. That is, the following is wrong and will result to nothing in this case:

```
window.addEventListener("load", loadGrid());
```

Now, what if, in the (near) future we want to call more than one function on window load? To do so, we simply create a wrapper function, as follows:

```
function windowOnLoad() {
    loadGrid();
}
```

```
window.addEventListener("load", windowOnLoad);
```

`windowOnLoad()` simply calls any functions we want to call once the window has been fully loaded (so far, only `loadGrid()`).

Have we forgotten something?

[Take time to think…]

As you might have guessed, since our browser will follow any instructions found in our `index.html` file when loading the page, we need to somehow inform it that there is a script it needs to load. To do so, we modify `index.html` as follows:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <link rel="stylesheet" type="text/css" href="./css/style.css" media="screen"/>
        <script type="text/javascript" src="./js/main.js"></script>
        <title>
            Minesweeper
        </title>
    </head>
    <body>
        <div id="grid-container" class="minesweeper-grid-container"></div>
    </body>
</html>
```

The key part above is the `<script>` tag, through which we inform HTML to look for our script and follow any instructions therein. Observe, also, how our HTML is quite shorter, since we have removed any redundant cell-specific code. (Re)loading `index.html` now should result to the very same layout as before.

Fun Time!

Placing Mines

One of the key parts of minesweeper are… mines. So, since we have a nice layout, we should now focus on adding some of the necessary game functionality, starting by placing mines and computing all those colourful numbers that should appear on the board.

Questions / Action Items:

1. What would be a reasonable way to represent the game's grid? Why?
2. How to place, say, 10 mines once the page has loaded? Write down some JS function(s) that implements your logic.

3. How to compute all those numbers that should appear on the game's board? Write down some JS that implements your logic.

Feel free to change anything on `./js/main.js` script to accommodate any new need that may occur.

Note: For the time being, let all numbers show up in their cells to help you in debugging. Also, use `-1` to denote mine position.

More Advanced Styling

As you might remember, all those numbers on the minesweeper grid are quite colourful and appear in a certain font. At this step, we are going to work on styling our cells a bit better.

Questions / Action Items:

1. How can we specify text font using CSS? Choose a nice font for our purpose.
2. How can we load a font from a local file? Why is this better?
3. Look up fontsquirrel and transfonter, which might come in handy for properly parsing a font.
4. Each number on the game's grid has a different colour. Does this mean that we need a different CSS class for each number or can CSS access a DOM element's content? Play around and implement the solution that fits you best.

Marking Mines

A nice feature of the original minesweeper UI is that one can right click on a cell to flag it as a (potentially) dangerous cell (i.e., one containing a mine).

Questions / Action Items:

1. Right click on a grid cell. What do you observe?
2. Is there a way to prevent the browser's default behaviour? Find out on the web!
3. Implement this feature. You might first need to find about the right event to listen to, in order to overwrite the browser's default behaviour on right clicks.

Clicking on Cells

So far, we have focused on some satellite features of the game, yet the most central part of the game is the mechanism that allows the player to click on a cell and uncover any underlying mines etc.

Questions / Action Items:

1. Play some games of minesweeper and make sure you understand which cells are uncovered each time you click a certain cell.

2. Once done, think of an algorithm that can implement the observed behaviour.
3. As you might have observed, when clicking a button, the cell's style changes, to indicate that it has been clicked / explored and you can no more click on it. Create any necessary CSS class(es) to capture the stylistic part of this behaviour.
4. Implement your algorithm using JS! Make sure that once you click on a cell you cannot click on it again.

Tick-Tack!

In the original minesweeper interface, there is a clock on the one corner of the interface and a score board on the other, measuring the number of remaining mines.

Question / Action Items:

1. Implement the score board, which counts the number of mines that remain to be found.
2. Implement the timer, which measures the number of seconds since the game has started.

Wrapping Up

Having implemented most of the required functionality, make sure once a game is over, the user is informed for its outcome, that they can start over again and anything else you find relevant.

Navigation and Footer

Action Items: 1. Preparing for the future, add a navigation bar and a footer at the end of the page. The navigation bar could contain links to the user's profile (or an option to log in), to their past scores / games history or anything else you find meaningful. Similarly, the footer could contain some credits and communication information.