Session 2: First Hands-On Coding Time

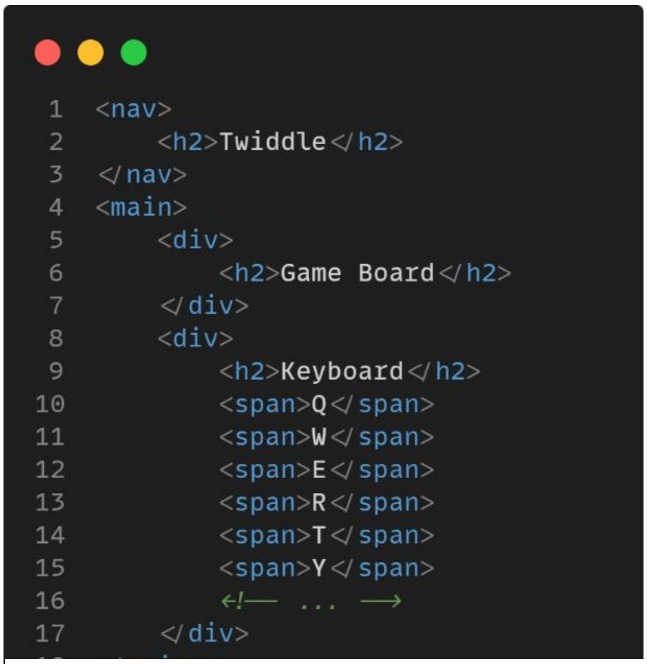
The code for this session is available at (tag: session-02):

https://github.com/LifeLongThinker/PracticalSoftwareDev.Twiddle

In which we learn:

- that keeping it simple is a best practice (**KISS** — Keep It Simple, Stupid);
- that **complex systems evolve from less complex ones**, so it's a good idea to start simple and iterate;
- that computers are so much better at executing **repetitive tasks** than human beings, so we should let them do the heavy lifting;
- that **naming stuff is hard** but necessary;
- that **symmetry** plays an important role in guiding the development process;
- that **separation of concerns** is another best practice;

- The first hands-on coding session is set against the background of our newly gained problem domain knowledge (last session). We know we will have to implemented the following parts, which will become part of our solution domain: **game board** with **tiles**, **keyboard** with **rows** of **keys**, **navigation** menu, **message** and **info box**.
- Setting out to implement our Wordle Clone we come up with the markup in *Listing 1*. We quickly realize two things:
    1. We are writing markup for multiple components

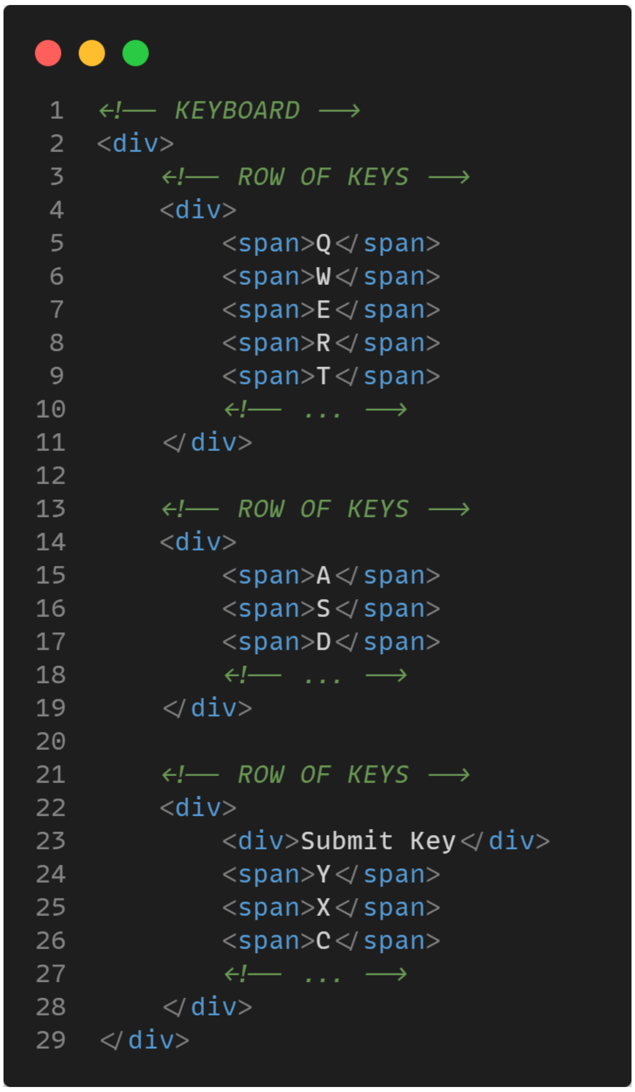


```html
<nav>
    <h2>Twiddle</h2>
</nav>
<main>
    <div>
        <h2>Game Board</h2>
    </div>
    <div>
        <h2>Keyboard</h2>
        <span>Q</span>
        <span>W</span>
        <span>E</span>
        <span>R</span>
        <span>T</span>
        <span>Y</span>
        <!-- ... -->
    </div>
```

Listing 1

   (navigation, game board, keyboard) from the very beginning and already using a wide range of specific tags. Both introduces needless complexity. Instead, we should opt for the **KISS** approach and start with something simpler, focusing on the keyboard component with standard tags (`div`/`span`) first. We can always evolve and use more specific tags as we go and necessity arises.
2. We might have to use a **more structured approach** here because we need to set the keys apart from each other in three rows to mimic the keyboard look that users know from visual keyboards. This is a semantic nuance that should be reflected in our (semantic) markup.

- Doing some refactoring, we have managed to reduce the markup to the bare essentials. The **semantics** clearly show now that we are dealing with a keyboard, three rows of keys, and keys (listing 2).
- Something else strikes us as odd: We have to do a lot of **boilerplate** coding. There is quite some **repetition** in our markup, which is both cumbersome to handle and can easily introduce errors.
- Instead of writing the markup for all the keys manually, we should use JavaScript to **have the computer write the markup for us**.
- Knowing that we can easily manipulate the DOM with JavaScript, we come up with the



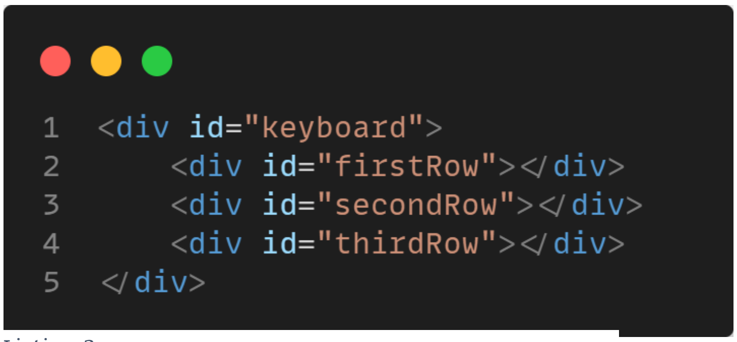
```html
<!-- KEYBOARD -->
<div>
    <!-- ROW OF KEYS -->
    <div>
        <span>Q</span>
        <span>W</span>
        <span>E</span>
        <span>R</span>
        <span>T</span>
        <!-- ... -->
    </div>

    <!-- ROW OF KEYS -->
    <div>
        <span>A</span>
        <span>S</span>
        <span>D</span>
        <!-- ... -->
    </div>

    <!-- ROW OF KEYS -->
    <div>
        <div>Submit Key</div>
        <span>Y</span>
        <span>X</span>
        <span>C</span>
        <!-- ... -->
    </div>
</div>
```

Listing 2

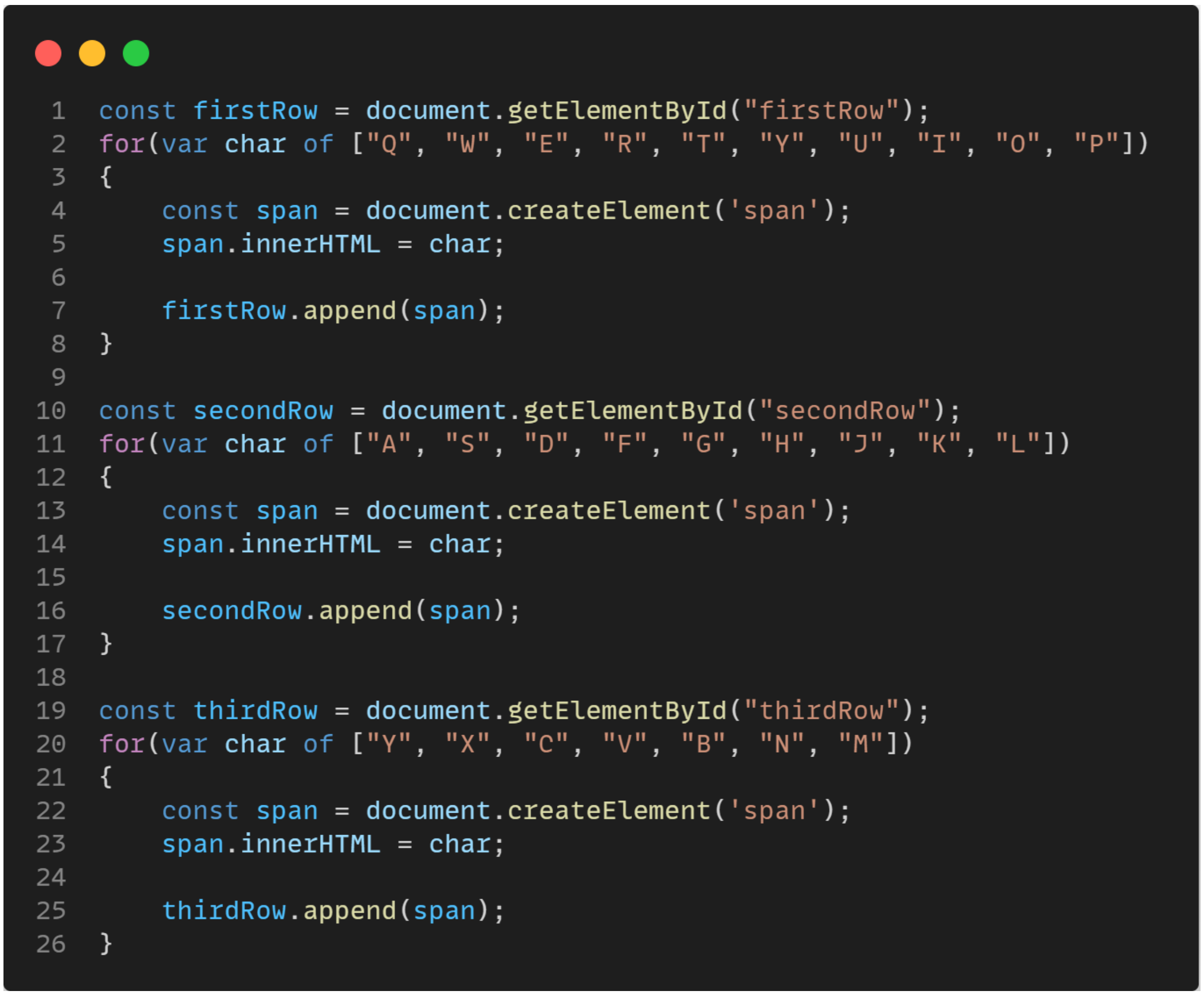
following markup skeleton and code to render the markup for the keys programmatically (listings 3 and 4).



```html
<div id="keyboard">
    <div id="firstRow"></div>
    <div id="secondRow"></div>
    <div id="thirdRow"></div>
</div>
```

Listing 3

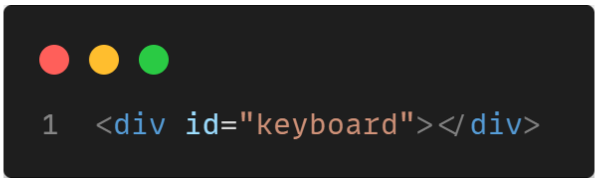
- This is definitely a step forward, but again there is a lot of repetition — only this time it's inside the JavaScript code, not the markup itself (cp. lines 1 to 8 with 10 to 17 and 19 to 26 in listing 4). As a rule of thumb, whenever we find ourselves including numbers in variable names or identifiers (cp. `firstRow`, `secondRow`, `thirdRow` in listing 3 and 4), this is a telltale sign that we are still doing too much repetitive work. Again, we model our logic in such a way to have the markup (this time of the rows of



```javascript
1   const firstRow = document.getElementById("firstRow");
2   for(var char of ["Q", "W", "E", "R", "T", "Y", "U", "I", "O", "P"])
3   {
4       const span = document.createElement('span');
5       span.innerHTML = char;
6
7       firstRow.append(span);
8   }
9
10  const secondRow = document.getElementById("secondRow");
11  for(var char of ["A", "S", "D", "F", "G", "H", "J", "K", "L"])
12  {
13      const span = document.createElement('span');
14      span.innerHTML = char;
15
16      secondRow.append(span);
17  }
18
19  const thirdRow = document.getElementById("thirdRow");
20  for(var char of ["Y", "X", "C", "V", "B", "N", "M"])
21  {
22      const span = document.createElement('span');
23      span.innerHTML = char;
24
25      thirdRow.append(span);
26  }
```

Listing 4

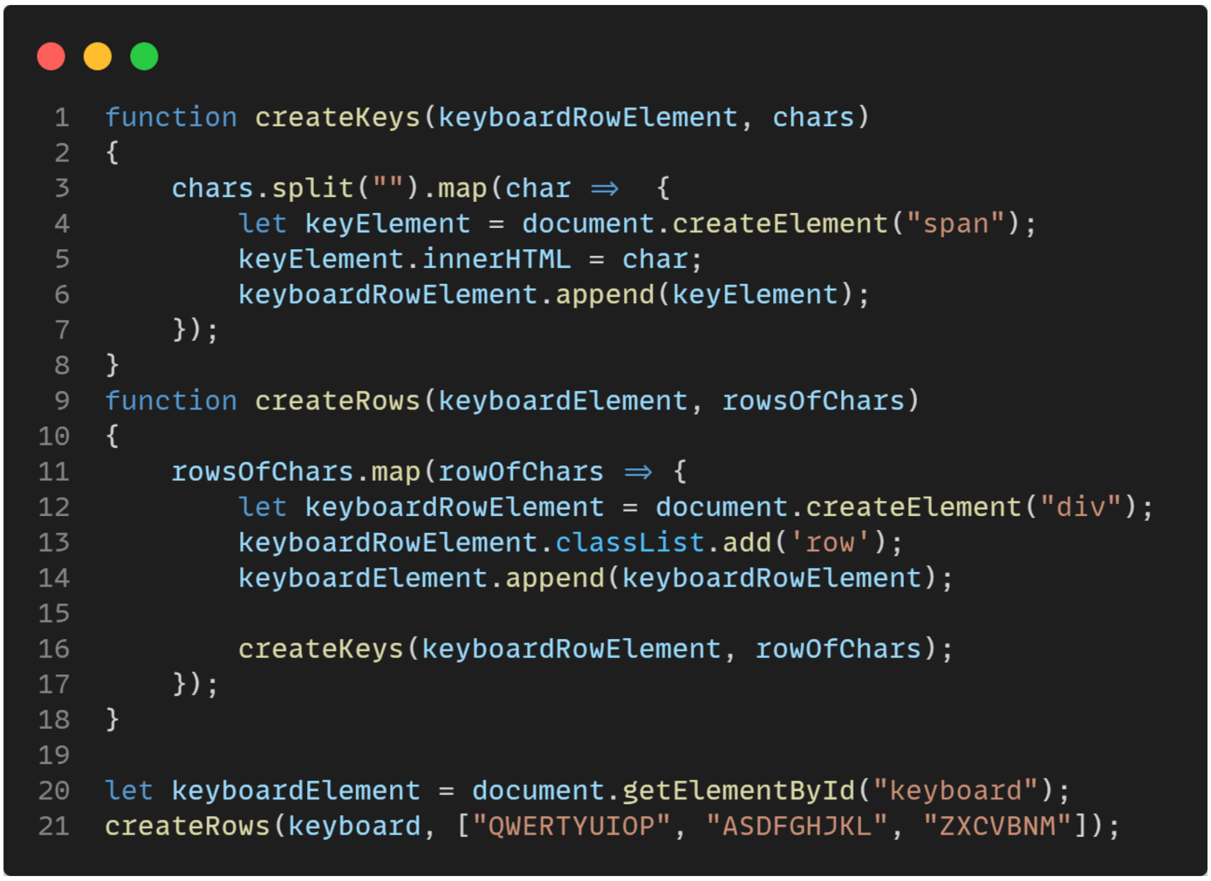
keys) generated iteratively. We also get rid of the arrays of strings (the key characters, cp. lines 2, 11, and 20 in listing 4) and instead split a string of key characters, which we then map() to have them inserted into the key row, which saves us the imperative `for` loop. This way we replace imperative-style code with declarative code that is typically used in functional programming.



```html
<div id="keyboard"></div>
```

Listing 5

- Finally, we end up with the code in listings 5 and 6.



```javascript
function createKeys(keyboardRowElement, chars)
{
    chars.split("").map(char =>  {
        let keyElement = document.createElement("span");
        keyElement.innerHTML = char;
        keyboardRowElement.append(keyElement);
    });
}
function createRows(keyboardElement, rowsOfChars)
{
    rowsOfChars.map(rowOfChars => {
        let keyboardRowElement = document.createElement("div");
        keyboardRowElement.classList.add('row');
        keyboardElement.append(keyboardRowElement);

        createKeys(keyboardRowElement, rowOfChars);
    });
}

let keyboardElement = document.getElementById("keyboard");
createRows(keyboard, ["QWERTYUIOP", "ASDFGHJKL", "ZXCVBNM"]);
```

Listing 6

- Some final remarks:
    - To style our keyboard, we are using CSS in a separate stylesheet (not shown here). This separation of UI and semantic markup is generally good practice and follows the **principle of separation of concerns**.
    - To apply styles from a separate stylesheet fine, we need to target the nodes we want to style. To do so, we inject class names and identifiers. It is very important to **use proper names** here.