TypeScript and the DOM

Tic Tac Toe

Let's create an interactive Tic Tac Toe game for two players.

As always, we will start with static HTML and CSS

```
<!DOCTYPE html>
<html>
  <head lang="en">
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Tic Tac Toe</title>
    <link rel="shortcut icon" href="/favicon.ico" />
  </head>
  <body>
    <h1>Tic Tac Toe</h1>
    <ul>
      <li></li>
      <li></li>
      <li></li>
      <li></li>
      <li></li>
      <li></li>
      <li></li>
      <li></li>
      <li></li>
      <li></li>
    </ul>
    <script type="module" src="/src/main.ts" charset="utf-8"></script>
  </body>
</html>
```

CSS Root Variables

```
:root {
  /* CSS Variables for all the font colors and sizes. Try changing these! */
  --header-background: #5661b3;
  --header-text-color: #fff9c2;
  --header-font-size: 2rem;
  --square-font-size: calc(8 * var(--header-font-size));
  --square-text-color: #5661b3;
  --square-background-color: #e6e8ff;
  --square-border: 3px solid var(--square-text-color);

  font: 16px / 1 sans-serif;
}
```

CSS html/body formatting

```
html {
  height: 100%;
}

body {
  margin: 0;
  min-height: 100%;
}
```

Style the header

```
h1 {
  /* center the header */
  text-align: center;

  /* Use a sans serif font with a little spacing and color */
  font-family: Verdana, Geneva, Tahoma, sans-serif;
  letter-spacing: 0.4rem;
  font-size: var(--header-font-size);
  color: var(--header-text-color);

  /* Remove margins and set a little padding */
  margin: 0;
  padding: var(--header-font-size);

  /* Set a background color for the header */
  background-color: var(--header-background);
}
```

Reset ul/li styles

```
ul,
li {
  /* Be gone margins! */
  margin: 0;
  padding: 0;

  /* and list styles */
  list-style: none;
}
```

Grid format the main ul

```
ul {
  /* Make the height of the list equal to the height of the page MINUS the height taken by the header */
  height: calc(100vh - 3 * var(--header-font-size));

  /* Display the list as a 3 column and three row grid */
  display: grid;
  grid-template: 1fr 1fr 1fr / 1fr 1fr 1fr;

  /* Add a little gap between to allow the background color through */
  gap: 1rem;

  /* Set the background color that will show through the gap */
  background-color: var(--square-text-color);
}
```

Style the li, in our case, the Tic Tac Toe cell

```
ul li {
  /* Use a monospace font */
  font-family: monospace;
  font-size: var(--square-font-size);

  /* Style the background color of the item */
  background-color: var(--square-background-color);

  /* Make the cursor a pointer by default */
  cursor: pointer;

  /* Center the text in the LI */
  display: flex;
  align-items: center;
  justify-content: center;

  /* Don't let the squares become too small */
  min-width: 3rem;
  min-height: 10rem;
}
```

Add styles for cells that are *taken* and *not-allowed*

```
ul li.taken {
  cursor: not-allowed;
}

ul li.small {
  font-size: 4rem;
}

ul li.not-allowed-click {
  background-color: red;
}
```

Let's add an "X" to one of our cells and mark it as taken

Could we dynamically change the content?

Time to make our page reactive by adding in TypeScript that interacts with the browser's content

TypeScript, when run in the browser, has access to the DOM (Document Object Model)

Remember that `Elements` view in the web developer's tool?

That isn't our `index.html` file! The browser's interpretation of the content into a nested set of *objects* the browser uses to *render* our page.

But wait! We know how to use `objects` in TypeScript!

If we had a way to access those *live* objects in our code, we could adjust them.

If we can adjust them, the browser would show us the updated view!

Enter the `DOM` and the TypeScript `DOM` API!

DOM API

Open up developer tools

We've seen some "built-in" objects already.

console

```
console.log('Wow!')
```

window

```
window.alert('Watch out!')
const name = window.prompt('Your name?')
```

What about interacting with the current "document"?

In the console, type: `document` and press enter

```
document
```

This document **is the *LIVE* view of our HTML code**

Our browser loads the HTML and creates objects for *each* element

The *DOM* (Document Object Model) is the browser's view of the live content

How do we get access to these elements so we can change them?

document gives us methods to start finding (`select'ing) elements.

So many options!

There are many functions the `document` can do, but `querySelector` is the most basic

`querySelector` Looks for the *FIRST* matching element

```
document.querySelector( ... css selector syntax as a string ...)
```

Example:

```
document.querySelector('li')
```

Will return the first `li` element in the document, scanning from top to bottom.

Gives back a *real* object, something we could assign to a variable!

This opens up an entire world!

We didn't get a `string` or a `number`; we received an `HTMLElement`

Dev tools show it formatted like HTML

- We can expand it to show all the element's attributes (Wow, so many!)
- We can hover it to highlight it in the browser window
- We can click on it to jump to that element in the Elements tab

Put it in a variable

```
firstListItem = document.querySelector('li')
```

While there are *many* attributes to investigate, let us start by looking to see what *text* is inside this element.

```
firstListItem.textContent
```

Since the `li` is empty, we get back an empty string!

Update the HTML code to put an `O' in the first li

Rerun the example, and we'll see this code return "0"

```
firstListItem = document.querySelector('li')
firstListItem.textContent
```

But that isn't all! We can *change* the `textContent`

```
firstListItem.textContent = 'X'
```

Whoa! The `O` became an `X`

We could even *add* to the text

```
firstListItem.textContent = firstListItem.textContent + '*'
```

or

```
firstListItem.textContent += '*'
```

But what about getting and changing attributes? (e.g. 'class')

Update the original HTML to put a taken class on that first LI.

Now in the console we can do:

```
firstListItem = document.querySelector('li')
firstListItem.className
```

Wow, we get back taken, which is the class name we assigned!

But what if there is more than one class on the item?

Enter `classList`

```
firstListItem = document.querySelector('li')
firstListItem.classList
```

Put *two* classes in the HTML

```
<li class="taken small">X</li>
```

Now see that the `classList` has two elements!

Manipulate the class list!

- We can ask the class list if it contains a class:

```
firstListItem.classList.contains('taken')
```

- We can also *remove* a class!

```
firstListItem.classList.remove('small')
```

- We can also *add* a class. We can put back the `small` class

```
firstListItem.classList.add('small')
```

- Since we add/remove classes so often, we can even `toggle` them on and off

```
firstListItem.classList.toggle('small')
```

Now, let us interact!

We can now:

- Find an element
- Get its content
- Change its content
- Add classes, remove classes, toggle classes

How can we interact with the element? Functions!

handleClickSquare

```
function handleClickSquare() {
  window.alert('You did it!')
}
```

Tell the `li`: Every time a click happens, here is what to do...

The browser is sitting there *waiting* for events to happen. The browser is event **driven**

We can teach the `firstListItem` object new tricks

```
//
// What object
//      |
//      |         please listen for an event
//      |              |
//      |              |           name of event to wait for
//      |              |              |
//      |              |              |      function to call when it happens
//      |              |              |         |
//      v              v              v         v
firstListItem.addEventListener('click', handleClickSquare)
```

```
firstListItem.addEventListener('click', handleClickSquare)
```

This means:

The list item in the variable `firstListItem` *should listen for a specific kind of event, in this case, a* `click`*, and when it happens, the browser will CALL BACK our* `handleClickSquare` *function*

There is a list of pre-defined events that the elements know!

It is a long list, but you don't need to know them all.

Try clicking on that element!

Let us try a different event: Mouse Move

```
function handleMouseMove() {
  console.log('moving!')
}

firstListItem.addEventListener('mousemove', handleMouseMove)
```

Improve handleClickSquare to make it *DO* something

Reload the page to wipe out our work and start clean.

```
firstListItem = document.querySelector('li')
```

Now we will redefine `handleClickSquare` but add a parameter (argument) to our function.

```
function handleClickSquare(event) {
  // Code goes here
}
```

Every time our browser handles an event

... we get our function called *AND* we get an event object!

Let us check out that event!

```
function handleClickSquare(event) {
  console.log(event)
}

firstListItem.addEventListener('click', handleClickSquare)
```

The `event` knows which element generated the click!

Check out the `target` property of the event!

It is the element we clicked on!

Rewrite `handleClickSquare` to change the text

Reload!

```
firstListItem = document.querySelector('li')

function handleClickSquare(event) {
  const thingClickedOn = event.target

  thingClickedOn.textContent = 'X'
}

firstListItem.addEventListener('click', handleClickSquare)
```

NICE!

However, how do we do this for *ALL* the list items?

document.querySelectorAll

```
const allSquares = document.querySelectorAll('li')
```

Try to write `allSquares.addEventListener`

❌❌❌❌❌

We can't *directly* call it

`querySelectorAll` returns a *NodeList*, so we have to loop...

`forEach` will work!

```
allSquares.forEach(square =>
  square.addEventListener('click', handleClickSquare)
)
```

The loop adds an event listener to each item!

Done playing in the console, to the editor and TypeScript!

Now that we have some experience let us put it in our actual TypeScript file!

Bring our code over to `main.ts`

```
import './style.css'

function handleClickSquare(event: MouseEvent) {
  // Get the target of the click
  const thingClickedOn = event.target

  thingClickedOn.textContent = 'X'
}

const allSquares = document.querySelectorAll('li')

allSquares.forEach(square =>
  square.addEventListener('click', handleClickSquare)
)
```

There are red-squiggles!

Telling TypeScript to check!

```
import './style.css'

// Defines a method for us to handle the click
function handleClickSquare(event: MouseEvent) {
  // Get the target of the click
  const thingClickedOn = event.target

  // If the thing clicked on is an LI Element
  // - This does several things:
  // - 1. Checks that the target isn't null
  // - 2. Let's TypeScript know that *inside* this if statement
  //      the thingClickedOn is an LI element, and thus we can
  //      change its textContent
  if (thingClickedOn instanceof HTMLLIElement) {
    thingClickedOn.textContent = 'X'
  }
}

const allSquares = document.querySelectorAll('li')

allSquares.forEach(square =>
  square.addEventListener('click', handleClickSquare)
)
```

Hover over thingClickedOn before and after the `if`!

TypeScript allows for *type narrowing*.

The more we tell/test the more TypeScript can know!

Load the page and see our logic works each time we reload the page!

But our game isn't great.

`X` keeps getting to take squares!

We should toggle between players!

Time for *STATE*

```
let currentPlayer = 'X'

function handleClickSquare(event: MouseEvent) {
  const thingClickedOn = event.target

  if (thingClickedOn instanceof HTMLLIElement) {
    thingClickedOn.textContent = currentPlayer
  }
}
```

Ok, but we need to change who's playing after each click!

Bring in some *behavior*

```
// If currentPlayer is precisely the text 'X', make the currentPlayer 'O'
if (currentPlayer === 'X') {
  currentPlayer = 'O'
} else {
  // Otherwise it was already 'O', so make it an 'X'
  currentPlayer = 'X'
}
```

But what if we made a mistake!

```
// If currentPlayer is precisely the text 'X', make the currentPlayer 'O'
if (currentPlayer === 'X') {
  currentPlayer = 'O'
} else {
  // Otherwise it was already 'O', so make it an 'X'
  currentPlayer = 'Y'
}
```

Make the definition of `currentPlayer` more specific!

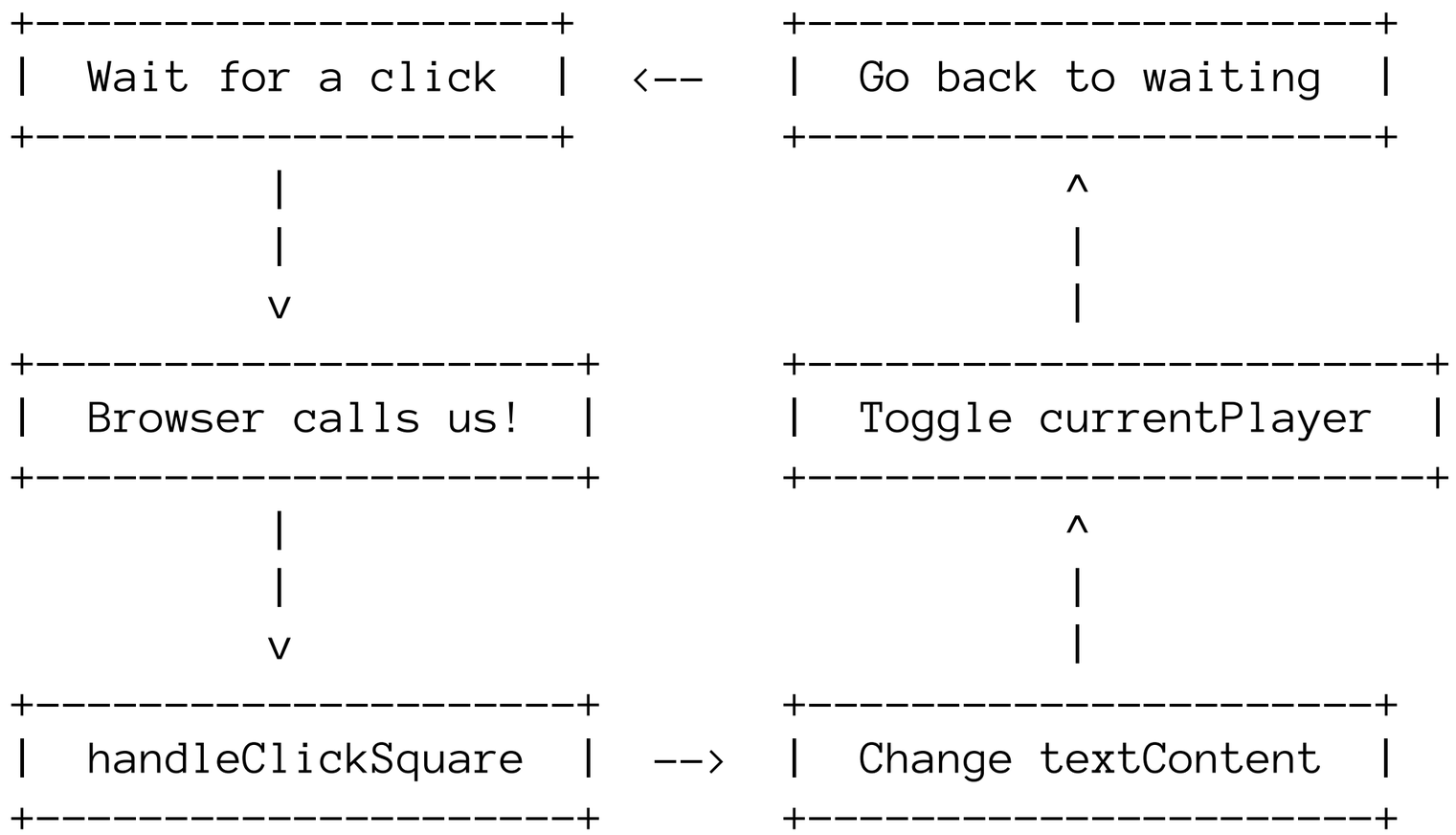
```
let currentPlayer: 'X' | 'O' = 'X'
```

This means that currentPlayer is a string, but can *only* be the string X or the string O

TypeScript will check to make sure this is always true in our code, but only during coding and compiling!

Get clicking!

What is happening?



Oooops, we can click a square twice!

First, let us mark the square as taken!

Add the taken class to an element.

```
thingClickedOn.classList.add('taken')
```

We can add logic/behavior to block clicking twice!

Before we do any work in `handleClickSquare`

Algorithm:

- Look at the `li`
- If the `li` contains the class `taken`
    - ... then do nothing, stop work!
- Otherwise, proceed...

Guard clause

```
if (thingClickedOn.classList.contains('taken')) {
  console.log('Nopes....')
  return
}
```

Except for winning/losing, we are looking pretty good!

Let us add a counter of the number of moves!

MORE *state*

```
let moveCounter = 0
```

Then every time we make a good move:

```
// Increment the move counter
moveCounter++
```

See in the console we can look at the `moveCounter` as we click.

Change the header

First, query for the element

```
// Get the header to query for the first `h1`
const header = document.querySelector('h1')

// Interpolate a string with the header and the count of moves
// and replace the text content of our header!
header.textContent = `Move ${moveCounter} of Tic Tac Toe`
```

What have we added to our tool belt?

- `document.querySelector`
  - Returns the first element that matches the given selector
- `document.querySelectorAll`
  - Returns ALL the elements that match the given selector
- `element.addEventListener`
  - Waits on a specific event to happen *to* a specific element

That is all!

With these simple tools, we can make any page dynamic.

... However, there are better tools ...

Bonus Content!
Event bubbling!

Remember when we had to add event listeners to *all* the `LI` to handle the clicks?

We did not know that the `click` on an `li` will move *up* the DOM tree to the parent, grandparent, and the great grandparent, hoping someone will be `listening` to that event.

Instead of `querySelectorAll('li')` **we can get the** `ul`

Delete the code relating to `allSquares` and replace it:

```
const gameBoard = document.querySelector('ul')

gameBoard.addEventListener('click', handleClickSquare)
```

But wait ...

When a `li` click happens, it will "bubble up" to the `ul`

Our `handleClickSquare` is waiting for it!

Since the `event.target` will be the clicked `li`, all our code will work!

It is possible to click on the `ul` itself. Click on the gaps between squares.

Luckily our instance on type safety helps us! We are already testing for clicks against an `<li>`!

Event bubbling can reduce the amount of code, but it can be tricky.