TypeScript

Goal

In this lab, you will:

- Create a RPN calculator using a TypeScript class
- Maintain a stack of numbers in an array
- Handle events from DOM objects
- Update the DOM with intermediate values

Your mission

This is lab you will use TypeScript to create a simple Reverse Polish notation (RPN) calculator. A RPN calculator consists of a stack of operands. The mathematical operators work on the last two operands on the stack. After performing the operation, the resulting value is pushed back onto the stack. For example: the formula `2 * (3 + 4)` is entered as `2 Enter 3 Enter 4 + *` and should result in `14`.

The RPN calculator takes input from the browsers DOM and update the DOM after each operation.

Note: You can read more about RPN on Wikipedia.

Type it out by hand?

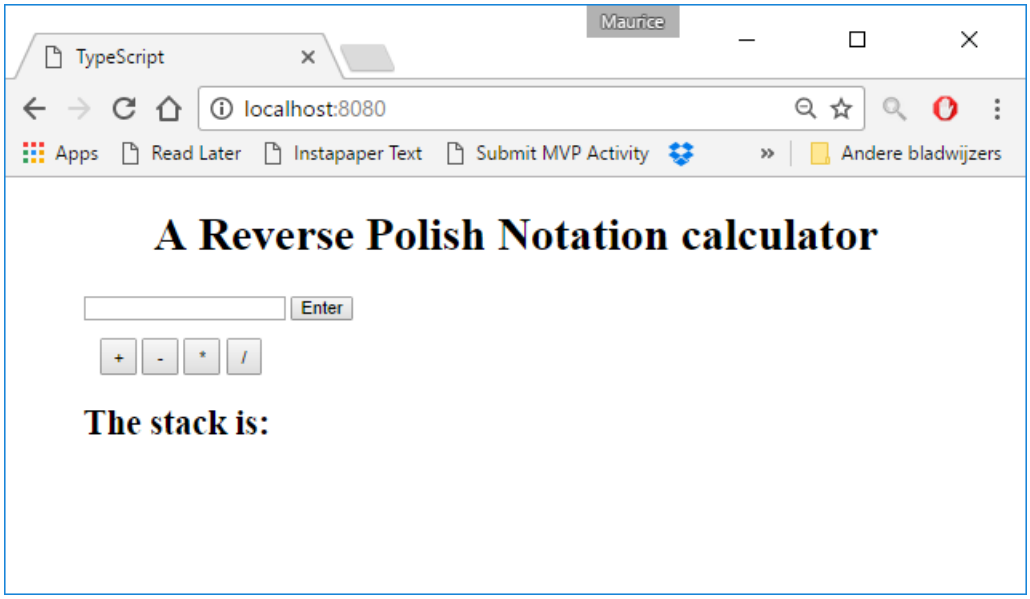
> Typing it drills it into your brain much better than simply copying and pasting it. You're forming new neuron pathways. Those pathways are going to help you in the future. Help them out now.

Create a basic calculator

In this section you will create the start of a basic RPN calculator class. The RPN calculator will accept inputs from the `<input>` element. It will display the content of the stack using an `<ul>` element.

1. Open the **begin** folder and observe that there are three files.
    1. The first is **index.html** and contains the markup needed for this exercise. This file is already complete and you do not need to make any changes to it.
    2. The second if **site.css** and it contains some simple CSS styling. This file is also complete and you do not need to make any changes to it.
    3. The third file is **index.ts**. This file contains the empty TypeScript class `RPNCalculator`. It is your job to add the code required for this exercise in this file.

2. Open the **index.html** in the browser of your choice. You should see the following page load in the browser. Please note that clicking any of the buttons has no effect yet. This is because the required functionality isn't implemented yet.



3. Open the **index.ts** file. Add a constructor to the `RPNCalculator` class. This constructor takes a parameter named `root` of type `Document`. Sore the root parameter in a private property with the same name. Create a new instance of this `RPNCalculator` type passing in the main document from the DOM.

```
(function () {
    "use strict";

    class RPNCalculator {
        constructor(private root: Document) {
        }
    }

    new RPNCalculator(document);
}());
```

4. One of the advantages of using TypeScript is that the **tsc** compiler can warn you of problems with you code. Open a console or terminal window and navigate to the begin folder. Compile the **index.ts** file to create an ECMAScript file browser can understand with the command `tcs .\index.ts`. You can compile the TypeScript whenever you want to but that is not convenient. A much nicer approach is to have the compiler watch for changes. You can do this with the command `tsc .\index.ts --watch`. If everything is well there should be no errors reported and an index.js file should be created. If the TypeScript compiler prints any errors in any of the steps you should fix these before continuing.
5. The markup contains an `<input>` element with id `number`. There is also an `<ul>` element with id `stack`. Add two private properties to the `RPNCalculator` class named `numberInput` and `stackElement`. Use these properties store the references of the two DOM elements.

```
class RPNCalculator {
    private numberInput: HTMLInputElement;
    private stackElement: HTMLUListElement;

    constructor(private root: Document) {
        this.numberInput = <HTMLInputElement>root.getElementById("number");
        this.stackElement = <HTMLUListElement>root.getElementById("stack");
    }
}
```

6. You will need to listen to the click event for the `<button>` elements. Add a function `addEventListeners()` to the RPNCalculator class and call this from the constructor. Additionally, set the focus to the `<input>` element to make it easier to start entering numbers.

```
constructor(private root: Document) {
    // Previous code

    this.addEventListeners();
    this.numberInput.focus();
}

addEventListeners() {
}
```

7. For an RPN calculator you will need a stack to hold all numbers used as operands. Add a `stack` property of type number array to the RPNCalculator class.

```
class RPNCalculator {
    private stack: number[] = [];
    // Previous code
}
```

8. When the user of the calculator clicks on the Enter button you should take the value property from the `<input>` element. Next convert this value to a number and push this into the **stack**. Add a `pushAndClearInput()` function to the RPNCalculator class to do this. Also add an event handler in the `addEventListeners()` function to call this whenever the Enter `<button>` is clicked.

```
addEventListeners() {
    this.root
        .getElementById("btnEnter")
        .addEventListener("click", () => this.pushAndClearInput());
}

pushAndClearInput() {
    const value = +this.numberInput.value;
    if (!isNaN(value)) {
        this.stack.push(value);
    }
    this.numberInput.value = "";
    this.numberInput.focus();
}
```

9. Each time a number is pushed onto, or popped from, the stack you need to display the current stack values to the user. Add a `displayStack()` function and call this after pushing a number onto the stack from the `pushAndClearInput()` function. In the `displayStack()` function you should first remove any old values. Next you should add a new `<li>` element for each value on the stack.

```
displayStack() {
    let child = this.stackElement.firstElementChild;
    while (child) {
        this.stackElement.removeChild(child);
        child = this.stackElement.firstElementChild;
    }

    for (const value of this.stack) {
        const liElement = this.root.createElement("li");
        liElement.textContent = value.toString();
        this.stackElement.appendChild(liElement);
    }
}

pushAndClearInput() {
    const value = +this.numberInput.value;
    if (!isNaN(value)) {
        this.stack.push(value);
        this.displayStack();
    }
    // Previous code
}
```

10. You should now be able to populate the calculators stack.
    1. First make sure there are no TypeScript compilations errors.
    2. Refresh the browser and make sure you can enter numbers and they are added to the stack when you click Enter.

Pop values from the stack and use them

Now you have a stack of numbers we can start execution operations on them.

1. All operations work on pairs of values from the stack. Add a `nextValues()` function that pops two values from the stack and returns when as an object with `x` and `y` properties. Keep in mind that the stack can become empty. In that case you should return the value zero.

```
nextValues() {
    return {
        x: this.stack.pop() || 0,
        y: this.stack.pop() || 0
    };
}
```

2. Add a `calculate()` function. This will be a higher order function that takes the actual calculation to perform as a function parameter. The `calculate()` function should push any remaining input onto the stack. Next it needs to call the `nextValues()` function to get the next two operands to perform the operation on. When done it should push the result back onto the stack and display the current stack.

```
calculate(fn: Function) {
    if (this.numberInput.value) {
        this.pushAndClearInput();
    }

    const {
        x,
        y
    } = this.nextValues();

    this.stack.push(fn(x, y));
    this.displayStack();
}
```

3. With the `calculate()` function in place you can add event handler to the **Add**, **Subtract**, **Multiply** and **Divide** buttons. These need to be added in the `addEventListeners()` function.

```
addEventListeners() {
    // Previous code
    this.root
        .getElementById("btnAdd")
        .addEventListener("click", () => this.calculate((x, y) => x + y));

    this.root
        .getElementById("btnSubtract")
        .addEventListener("click", () => this.calculate((x, y) => x - y));

    this.root
        .getElementById("btnMultiply")
        .addEventListener("click", () => this.calculate((x, y) => x * y));

    this.root
        .getElementById("btnDivide")
        .addEventListener("click", () => this.calculate((x, y) => x / y));
}
```

4. You should now be able to perform calculations with the calculator.

    1. First make sure there are no TypeScript compilations errors.
    2. Refresh the browser and make sure you can use it to perform calculations. For example: the formula **`2 * (3 + 4)`** can be entered as **`2 Enter 3 Enter 4 + *`** and should result in **`14`**.

Bonus Exercise: Add keyboard support

Finished with the previous steps? Still have time left? Then consider this bonus assignment.

The RPN calculator is now fully functional but the usability leaves a bit to be desired. Using a calculator with just the keyboard is a lot more productive then having to switch between keyboard and mouse input. In this bonus exercise you will add keyboard only support to the RPN calculator.

1. Add an event handler for the `keypress` event to the `<input>` element. In the event handler check the `keyCode` property of the `e` parameter to determine what action to perform. When the user pressed Enter add the current value to the stack. When the user presses one of the **Add**, **Subtract**, **Multiply** or **Divide** keys perform the same action as the corresponding buttons do.

```
addEventListeners() {
    // Previous code

    this.numberInput.addEventListener("keypress", e => {
        switch (e.keyCode) {
            case 13:
                this.pushAndClearInput();
                break;
            case 42:
                this.calculate((x, y) => x * y);
                e.preventDefault();
                break;
            case 43:
                this.calculate((x, y) => x + y);
                e.preventDefault();
                break;
            case 45:
                this.calculate((x, y) => x - y);
                e.preventDefault();
                break;
            case 47:
                this.calculate((x, y) => x / y);
                e.preventDefault();
                break;
        }
    });
}
```

Solution

The Solution can be found in **complete** folder