Week 2 - Simulating a digital circuit

- Build familiarity with JavaScript for the assignment
- Work with JavaScript object's (prototype)
- Work with building blocks (wires) (logic gates) to build abstractions
- Have enough confidence in JS before diving into TypeScript (a superset of JS)

This week's tutorial will walk you through building a digital circuit simulator. This program will build your confidence in the JavaScript environment and hopefully clarify a lot of the syntax that will be necessary to read TypeScript fluently. Because an intimate knowledge of JavaScript is not the focus of this worksheet, a few helper functions have been provided in the **helper.js** file.

This worksheet is accompanied by a file called **worksheetChecklist.html**. With this file open in the browser, your code will be automatically checked. This contains tests for all the exercises contained in the worksheet.

You'll start by implementing code shown in the lecture and getting your first green ticks in the worksheetChecklist. Then you'll start building up the digital simulator.

We'll start by creating a "scheduler" that can keep a list of functions and the time in the simulation when they'll be run. We'll code it simply as a naive Priority Queue. A priority queue is a queue which returns the element with the highest priority. In our simulation, the highest priority will be the element with lowest simulation time. Once this is implemented you'll have invented simulated time!

Next you'll create wires. They'll be very simple digital wires that contain either a 1 or 0. Using functions they'll be able to pass on their signal or modify their signal. Modifying these functions is how you'll create logic gates and have invented a simple language of circuits in JavaScript.

The techniques taught in this worksheet can be built on in other contexts:

- Chat rooms (passing around chat messages)
- Turn based multiplayer games (passing around turns)
- Constraint propagation (passing around constraints)
- Passing around something!

Exercise 1 - Passing your first tests!

After you've downloaded the Tutorials code, open the file **worksheetChecklist.html** in the *Chrome* browser. This file tests your code and shows if the code works as expected. It only tests some cases.
There's a lot of red, so let's pass that first test!

Open **main.js**. This is where your code goes for each exercise. Notice you've been provided with `const myObj = undefined;`.

Please replace `undefined` with an object. This *myObj* object needs to be similar to the *myObj* shown in the notes.
It needs a key *aProperty* with any string as a value and a key *anotherProperty* with any number as a value. (Opposite to what's in the notes)

Once you've written an answer into the **main.js** file, save your file and *refresh* the browser. If the browser doesn't change, try hard refreshing (which makes sure your file is reloaded into the browser and not from cache).

Exercise 2 - C => Closures

Functions can be written in many different ways. In the lecture you've seen that the following is possible:

```
function add(x) {
    return y => y+x;
}
let addNine = add(9)
addNine(10)
> 19
addNine(1)
> 10
```

Here is another identical way of writing the **add** function from above.

```
const add = x => y => x + y
```

Here we're passing numbers into **add**, but it's also possible to pass functions as arguments. Please create a function **operationOnTwoNumbers** that takes a function as its argument and returns a function that must be called twice (like the **add** function above) before returning an answer.

A concrete example will make it easier to understand. We want to be able to do the following:

```
const add = operationOnTwoNumbers((x,y) => x + y)
const addNine = add(9)
addNine(3)
> 12
const multiply = operationOnTwoNumbers((x,y) => x * y)
const double = multiply(2)
double(4)
> 8
```

It works when all the green ticks appear!
Note: If you want to write a block of code in an arrow function you can use curly braces, i.e.

```
let sayName = name => {
    let message = "Hello ";
    console.log(message + name);
}
```

Exercise 3 - A Brief Look at Arrays

In the lecture you've seen the following:

```
['tim', 'sally', 'anne'].forEach(person=> console.log('hello ' +
person))
```

You'll need a function that calls all the functions stored in an array. This should be quite a small function. Please write a function **callEach(array)** which takes in an array of functions, and iterates along the array calling each function. Don't use the JavaScript loop construct, and instead use the **.forEach** method on the array.

Example:

```
const fn1 = _ => console.log("Hey!");
const fn2 = _ => console.log("Cool!");
callEach([fn1, fn2]);
> "Hey!"
> "Cool!"
```

Exercise 4 - Inventing Time

It's time to start building the simulation. We'll start with the most important part, the queue/scheduler/action dispatcher or **UniversalClock**. This epic sounding clock will keep time in our simulation allowing for delays to happen in function calls. It'll work like this:

```
let clock = new UniversalClock();
clock.addToSchedule(2, () => console.log("World!") );
clock.addToSchedule(1, () => console.log("Hello") );
runSimulation(clock);
> "Hello"
> "World!"
```

The UniversalClock constructor function has been given to you.

- *this.schedule* is the array where the [time, function] pairs will be stored.
- *this.simulationTime* will store the current time of the simulation.

this.schedule when containing entries should look something like this:

```
[ [2, fn1], [2, fn3], [10, fn2], [12, fn4] ]
```

It's important that the first field is the simulationTime when the function will trigger and not just the delay. Therefore, when implementing **addToScheduler** make sure to add the current simulationTime to the delay before working out where in the array to insert the [time, function].

Please implement the following:

a) Add the method **isEmpty** to the prototype **UniversalClock** that returns true if the array of actions is empty and false otherwise.
b) Add the method **addToSchedule** to the prototype **UniversalClock** that takes two arguments. A delay (number) and a payload (function). **addToSchedule** adds the function to the correct place in the array using the time when the function will be executed (time when scheduled + delay). The UniversalClock's array of actions should always be in ascending time order. It's also important that functions scheduled to happen at the same time are ordered in the order they were scheduled.
    - Use the helper function **insertInArray(array, index, item)** which returns a new array with the inserted item.
    - Argument 1 is the **delay** before the action function will be called.
    - Argument 2 is the **action** function to be called.
    - Please use the tests to find errors and see expected behaviour.

Exercise 5 - Experiencing Time

Now that we can add actions to the UniversalClock we need a method **getFirstItem** that returns the function that had been stored, and updates the UniversalClock simulation time to the time on that item.
If the scheduler is empty, throw an error:

```
throw new Error("UniversalClock is empty -- UniversalClock.getFirstItem");
```

Because the array is in order, use JavaScript's array method *shift* to remove the first item.

Please implement **getFirstItem**.

With these methods implemented we have invented time! Well not quite. We need to be able to actually run time. Here's the **runSimulation** function. Feel free to implement it yourself if you want a challenge. It just needs to grab the first item from the UniversalClock, call it, and repeat until the list is empty.

```
function runSimulation(clock) {
   if (clock.isEmpty()) {
       return;
   }
   clock.getFirstItem()();
   return runSimulation(clock);
}
```

Now we can test the UniversalClock concretely:

```
let clock = new UniversalClock();
clock.addToSchedule(2, _=> console.log("World!") );
clock.addToSchedule(1, _=> console.log("Hello") );
runSimulation(clock);
> "Hello"
> "World!"
```

Optional: Exercise 6 - Wires

Wires are just an object that contain a signal value and list of actions. Your Wire will only need two methods, **setSignal(signalValue)** and **addAction(actionFunction)**. **addAction** adds a function to an array of functions that will simulate signal passing. If we want a wire to send its signal to another wire, you can write the following function:

```
// This example doesn't use your scheduler yet.
function wirePassesSignal(inputWire, outputWire) {
    inputWire.addAction(() => outputWire.setSignal(inputWire.signalValue));
}
```

These two methods provide an incredibly powerful interface.

Whenever a wire's signal changes, all of its actions need to be called (Use **callEach** from Ex.3.). This propagates the change to the next wires. If the signalValue doesn't change, don't call the actions.

Whenever an action is added, it should be added to the *actions* array and called (to propagate the wires signal down this new connection).

Please satisfy the Wire's tests.

To help inspect your Wire, feel free to use the **probe** function contained in the helper.js file.

Optional Exercise 7 - Completing the Circuit language

With the Wire's methods **setSignal** and **addAction** we can now simulate our logic gates. Logic gates are functions that add the appropriate actions to the wires. For example, a **not** logic gate will be a function **not(wireIn, wireOut)** that adds an action to the **wireIn** to change the signal of **wireOut** to be the opposite of whatever its signal is! And we can include a delay if we add the setSignal method on the scheduler. To include our UniversalClock in the logic gate functions, we'll need to use the closure concept from Exercise 2.

The code below creates 3 functions.

- **addAfterDelay** - a function that takes in an instance of the UniversalClock and returns a new function with the clock hidden inside. This is the function we'll be passing into our logic gate's factory[2] functions.
- **logicalNot -** switches a 1 into a 0 or a 0 into 1.
- **notFactory** - takes in a delay and an afterDelayFn which will be the function returned by addAfterDelay.

Passing in functions makes the code more testable and less reliant on the global scope.

```
function addAfterDelay(clock) {
   return (delay, action) => clock.addToSchedule(delay, action);
}
const logicalNot = signal => signal === 0 ? 1 : 0;

const notFactory = (delay, afterDelayFn) => (input, output) => {
   const notAction = () => {
       afterDelayFn(delay, () => output.setSignal(logicalNot(getSignal(input))))
   }
   input.addAction(notAction);
}
```

The notFactory generates a function that takes an input wire and output wire. Passing in these wires causes the input wire to add an action which adds a function on the scheduler. Finally, when the scheduler triggers the function, the output wire has its signal changed to be the negation of the input wire. The notFactory also allows us to specify the delay in the not logic gate.

Here's the code for an **andFactory** and please implement an **orFactory**.

```
const andFactory = (delay, afterDelay) => (in1, in2, output) => {
```

[2] The Factory name references that the function creates other functions. notFactory is a function that *constructs* and returns a **not** logic gate function.

```
    const andAction = () => {
        afterDelay(delay, () => {
            output.setSignal(getSignal(in1) & getSignal(in2));
        });
    }
    in1.addAction(andAction);
    in2.addAction(andAction);
}
```

Challenge exercise - Half Adder

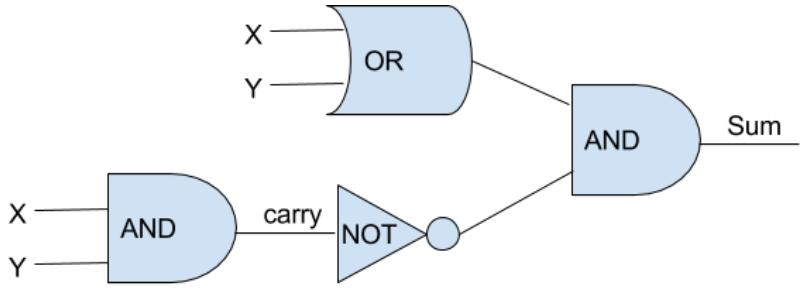
The final exercise requires you to implement a half adder. A half adder adds two single digit binary numbers together, and returns a sum and a carry. A half adder takes in two wires (**x** and **y**) and outputs on two wires (**sum** and **carry**). The truth table for a half adder is as follows:

x	y	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Using only the logic gates written, create a half adder function. It'll need to be a factory function that takes in an **or**, **not**, and **and** logic function.

```
const halfAdderFactory = (or, and, not) => (x, y, s, carry) => {/*YOUR CODE*/}
```

The [circuit diagram for a half adder] is also helpful:



Once you've implemented the half-adder, you can use it to construct (details here):

- A full adder
- A ripple carry adder