Stopwatch

A countdown timer with 'start', 'pause' and 'reset' functionality.

Exercise:

Create a stopwatch that counts down from a given number of seconds in the format `mm:ss`. Make it possible to `start`, `pause`, and `reset` the countdown. Make sure you can pass a callback function to the timer that is called when the displayed value is updated.

Solution:

The main part of this exercise boils down to modeling. If you find the right model, your life will be easy. If you use the wrong model, implementation will not only be hard, but the stopwatch might not reflect reality. Let's define the state space of the application:

- `countdownInitialValue`: the number of seconds initially set on the timer when instantiating it. When we reset the timer, its value will be set to this value.
- `secondsLeft`: the current value of the countdown timer in seconds

How do we make time pass?

The easy part of the answer is that we need to use the `setInterval` function.

`setInterval( callback, delay )` executes callback every `delay` milliseconds. Therefore, in theory, we could come up with the following solution:

```
setInterval( () => {
    this.secondsLeft -= 1;
    if ( this.secondsLeft === 0 ) { ... }
    // ...
}, 1000 );
```

Surprisingly, this approach may be highly inaccurate. When resources are scarce, the `setInterval` call is delayed. Small delays add up throughout the ten-minute countdown. Imagine that you do a resource-intensive task, and your browser might completely lose focus and priority in the background. It may happen that in five seconds, your callback function is only called twice, resulting in two-second progress instead of five.

If you want accurate results, you can make use of the fact that `Date.now()` returns the current timestamp, yielding the number of milliseconds passed since January 1st, 1970. The battle plan is as follows:

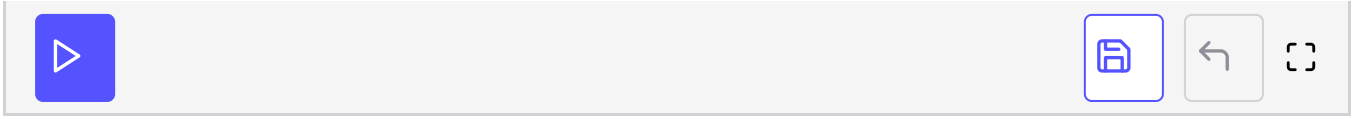
- We record the `startTimestamp` when we start the clock.
- Whenever we are inside the `setInterval` callback, we retrieve the current timestamp and update `secondsLeft`.
- The more often we run the `setInterval` call, the faster our results get updated. However, the results are independent of the `delay` value used in the `setInterval` call.
- We pause the countdown by terminating the `setInterval` call.
- We resume at `secondsLeft` once we continue the countdown. Note we might ignore up to 999 milliseconds of elapsed time with this approach.
- When resetting the counter, we equate `secondsLeft` to the initial value supplied in the constructor.

We will create an ES6 class to encapsulate all the necessary operations of the stopwatch:

```
class Timer {
    constructor( countdownInitialValue, displayTimeCallback ) {
        this.countdownInitialValue = countdownInitialValue;
        this.secondsLeft = countdownInitialValue;
        this.interval = null;
        this.displayTimeCallback = displayTimeCallback;
        this.displayTimeCallback( this.toString() );
    }
    toString() {
        const minutes = Math.floor( this.secondsLeft / 60 );
        const seconds =
            ( '' + ( this.secondsLeft % 60 ) )
                .padStart( 2, '0' );
        return `${minutes}:${seconds}`;
    }
    start() {
```

```
        let startTimestamp = Date.now();
        let startSeconds = this.secondsLeft;
        this.interval = setInterval( () => {
            let oldSecondsLeft = this.secondsLeft;
            let secondsPassed =
                Math.floor( ( Date.now() - startTimestamp ) / 1000 );
            this.secondsLeft =
                Math.max( 0, startSeconds - secondsPassed );
            if ( this.secondsLeft < oldSecondsLeft ) {
                this.displayTimeCallback( this.toString() );
            }
            if ( this.secondsLeft == 0 ) {
                clearInterval( this.interval );
            }
        }, 100 );
    }
    pause() {
        if ( typeof this.interval === 'number' ) {
            clearInterval( this.interval )
        }
    }
    reset() {
        this.pause();
        this.secondsLeft = this.countdownInitialValue;
        this.displayTimeCallback( this.toString() );
    }
}

const timer = new Timer( 61, console.log );
```



The constructor initializes the state of the application and sends the initial value to the callback. The `toString` method converts the `secondLeft` property to `mm:ss` format, making sure that `ss` has a leading zero. Notice the ES2017 `padStart` function.

The `start` method implements the stopwatch algorithm. Both the `pause` and the `reset` method stops the clock, and `reset` moves the application state to its initial value. When we press `reset`, we also have to call the `display` callback once.

You may think we are done with the exercise. Are we?

In every good exercise, there is a twist. Before reading any further, find a bug in the above implementation!

.

.

.

.

Welcome back! If you haven't done the exercise, I encourage you to go back and do it, because this is how you get better.

If you are still reading, congratulations, you have found a bug!

My solution is that after starting the clock twice and pausing it, the clock is still running. We will, therefore, guard the `start` method in the following way: if `this.interval` is a number, we will not start the clock, as it is already started:

```
start() {
  if ( typeof this.interval === 'number' ) {
    return;
  }

  let startTimestamp = Date.now();
  let startSeconds = this.secondsLeft;
  this.interval = setInterval( () => {
    let oldSecondsLeft = this.secondsLeft;
    let secondsPassed = Math.floor( ( Date.now() - startTimestamp ) / 1000 );
    this.secondsLeft = Math.max( 0, startSeconds - secondsPassed );

    if ( this.secondsLeft < oldSecondsLeft ) {
      this.displayTimeCallback( this.toString() );
    }

    if ( this.secondsLeft == 0 ) {
      clearInterval( this.interval );
    }

  }, 100 );
}
```

The `reset` method executes `pause`, so we have to make sure that `pause` sets `this.interval` to a non-numeric value after clearing the interval. Therefore, we have to add a line to set `this.interval` to `null`.

```
pause() {
  if ( typeof this.interval === 'number' ) {
    clearInterval( this.interval );
    this.interval = null;
  }
}
```

Now we can test the code and it works even if we started the clock twice.

Now we can test the code and it works even if we started the clock twice.