Stacks

introduction to stacks and stack size limits in Javascript

We will need a basic understanding of how stacks work to understand the next section. If you have never heard of a stack data structure, I will summarize it in this section. I still encourage you to do more thorough research before continuing, as you will need it in your programming career.

Imagine a stack like a JavaScript array whose elements are not accessible. Suppose stack S is given. You can only execute the following operations on it:

- `S.length`: checks the length of the stack
- `S.push(element)`: pushes an element to the stack
- `S.pop()`: removes the top element from the stack and returns it

You can neither access nor modify any elements of the stack other than the element at position `S.length - 1`. In Exercise 1, you will have a chance to implement a stack in ES6.

There are two types of memory available to you: the **stack** and the **heap**. The heap is used for **dynamic memory allocation**, while the stack is used for **static memory allocation**. Accessing the stack is very fast, but the size of the stack is fixed.

A *stack frame* is created for the global scope. Then, for each function call, another stack frame is added to the top. These frames stack on top of each other.

When executing JavaScript code, you get a stack with limited size to work with. To get an idea of typical stack size limits in practice, look at this page.

Regardless of the browser, iterating from zero to a million in a for loop is an easy task.

```
console.time( 'for loop' );
let sum = 0;
for ( let i = 0; i < 1000000; ++i ) {

    sum += i;
}
console.timeEnd( 'for loop' );
// for loop: 119.11ms
```

Doing the same iteration as a recursive call is an arduous task that most browsers are not able to solve because of the stack limit. Execution is fast, but the stack limit will not make it possible to execute a million calls.

```
function sumToN( n ) {
    if ( n <= 1 ) return n;
    return n + sumToN( n - 1 );
};

console.time( 'recursion' );
console.log( sumToN( 1000000 ) );
console.timeEnd( 'recursion' );
```

```
> Uncaught RangeError: Maximum call stack size exceeded...
```

Let's examine how the following program is executed on the stack:

```
function sumToN(n) {
    if ( n <= 1 ) return n;
    return n + sumToN( n - 1 );
};

console.log(sumToN(2));
```

1. Initially, a reference to the `sumToN` function is created on the global stack frame.

2. Once `sumToN(2)` is called, another stack frame is created, containing the value of `n`, the expected return value, and a reference to `sumToN`.

3. Once `sumToN(1)` is called, another stack frame is created with another

value of `n`, the expected return value, and a reference to `sumToN`.

4. Once `sumToN` returns `1`, the last stack frame is destroyed.
5. Once `sumToN(2)` is computed, the stack frame created in step (2) is destroyed.

Beyond a five digit stack limit, execution stops, and a JavaScript error is thrown. We will combat this behavior with tail call optimization.