Exercise on Tail Call Optimization and other Function Features in ES6

In the following exercises, you will use tail call optimization, create a stack, and examine how new.target behaves in ES5.

Exercise 1:

Implement a stack in ES6. In addition to a constructor, it should have a `push()`, `pop()`, and a `len()` function that would return the number of elements in the stack.

```
class Stack {
        //Write your code here
}
```

Exercise 2:

Write a tail call optimized solution for the following Fibonacci function. See if all of the tests work after you tail-call optimize the given function. Remember to name your function `fib2()` or the tests won't work.

```
function fib( n ) {
    if ( n <= 1 ) return n;
    return fib( n - 1 ) + fib( n - 2 );
}

function fib2(  )//add parameters
{
  //write code here
}
```

Explanation:

To avoid mixing the two Fibonacci functions, we will refer to the tail recursive fib function as fib2. We have to create accumulator variables to create proper tail calls. As we need to memorize the last two values of the sequence, we have to create two accumulators.

The accumulators will keep track of the last two elements that are needed for constructing the current Fibonacci number. Notice that the two accumulator values require us to create two separate exit conditions:

- When calling fib2 with 0, the return value should be 0.
- When calling fib2 with a positive integer, the final return value in the last fib2 call is acc1.

Example:

- `fib2(5)` becomes `fib2(5, 1, 0)` once the default values are assigned to the second and the third arguments
- `fib2(5, 1, 0)` invokes `fib2(4, 1, 1)`
- `fib2(4, 1, 1)` invokes `fib2(3, 2, 1)`
- `fib2(3, 2, 1)` invokes `fib2(2, 3, 2)`
- `fib2(2, 3, 2)` invokes `fib2(1, 5, 3)`
- as `n` is 1 for `fib2(1, 5, 3)`, 5 is returned as a result. 5 is the value of `acc1`.
- 5 is returned by all recursive calls. Therefore, `fib(5)` also returns 5.

Obviously, this implementation only works with non-negative integer inputs. We could also add another exit condition to handle invalid inputs:

```
if (n < 0 || !Number.isInteger( n ) ) return NaN;
```

Exercise 3:

Create a solution for the Fibonacci exercise that does not use recursion. Remember to name it `fib()`

```
function fib( n )
{
  count++;
```

```
  /*write-your-code-here*/

}
```

Exercise 4:

Rewrite the following code without using the class syntax of ES6. Observe how `new.target` behaves.

> Uncaught Error: Abstract class cannot be instantiated.(...)

```
class AbstractUser {
    constructor() {
        if ( new.target === AbstractUser ) {
            throw new Error( 'Abstract class.' );
        }
        this.accessMatrix = {};
    }
    hasAccess( page ) {
        return this.accessMatrix[ page ];
    }
}

class SuperUser extends AbstractUser {
    hasAccess( page ) {
        return true;
    }
}

let su = new SuperUser();

//let au = new AbstractUser();
// ^ Throws the new error
```