**API of the Tic-Tac-Toe module**

**Properties:**

* The game object (TicTacToe) has a property field, which represents the entire game field (3 x 3 fields) as a one-dimensional number array.
  + 0: not set
  + 1: set by player 1
  + 2: set by player 2
* Furthermore a property currentPlayer, which contains the current player (number 1 or 2).
* Last but not least there is a property gameState which contains the current state of the game:.
  + The game has four states: "open", "player1won", "player2won" and "draw".
  + The game has the state "open" as long as not all fields are occupied and nobody has won.
  + The game has the state "player1won" or "player2won", if the respective player could occupy three fields in a row (vertically, horizontally or diagonally).
  + The game has the state "draw" if all squares are occupied but nobody has won.

**Methods:**

* The game object has a method resetGame, which initializes a new game.
  + The method resets the game completely to the initial state.
* The game object has a method act, which takes as parameter the index (0 - 8) of the next game turn.
  + If the field does not exist (index invalid) an error with message "Invalid field index INDEX" (INDEX = the passed index) is thrown.
  + If the field is already occupied or the game is no longer in the open state, the method refuses the move and returns false.
  + If the move was successful, the method returns true.
  + After each successful move, the game state is updated and the current player is changed.
* To simplify manual testing on the console, helper methods can still be implemented.
  + For example, formatted output of the game field or the current game state.

Example usage

// game.js

const TicTacToe = require('./tic-tac-toe');

const ttt = new TicTacToe();

console.log(ttt.gameState); // "draw"

ttt.act(0); // Player 1

ttt.act(3); // Player 2

ttt.act(1); // Player 1

ttt.act(4); // Player 2

ttt.act(2); // Player 1

/\* Spielstatus: 1 1 1 <-- Gewonnen!

2 2 0

0 0 0 \*/

console.log(ttt.gameState); // "player1won"

ttt.resetGame();

console.log(ttt.gameState); // "open"

ttt.act(4); // Player 1

**Test**

Our modular design allows us to write unit tests! Please do this for your tic-tac-toe module. Since we write vanilla JS, we also use vanilla tests with node assertions :-).

Here are some examples to get you started:

const assert = require('assert');

const TicTacToe = require('./tic-tac-toe');

console.log('Testing TicTacToe...');

(function test\_Creation() {

assert.ok(new TicTacToe());

})();

(function test\_getGameState\_shouldInitiallyBeOpen() {

assert.equal(new TicTacToe().gameState, 'open');

})();

(function test\_currentPlayer\_shouldInitiallyBePlayerOne() {

assert.equal(new TicTacToe().currentPlayer, 1);

})();

(function test\_act\_shouldReturnTrueIfMoveIsValid() {

const ttt = new TicTacToe();

assert(ttt.act(0), 'act returned false');

})();

(function test\_act\_shouldRefreshGameState\_player1Won() {

const ttt = new TicTacToe();

ttt.act(0); // Player 1

ttt.act(3); // Player 2

ttt.act(1); // Player 1

ttt.act(4); // Player 2

ttt.act(2); // Player 1

assert.equal(ttt.gameState, 'player1won');

})();

(function test\_fields\_shouldChangeWhenPlayersAct() {

const ttt = new TicTacToe();

ttt.act(0); // Player 1

ttt.act(1); // Player 2

ttt.act(8); // Player 1

assert.deepEqual(ttt.fields, [1, 2, 0, 0, 0, 0, 0, 0, 1]);

})();

console.log('OK');

Supplement them with your own tests. If you want, before implementation (TDD).

**Bonus**

* Implement a simple AI that plays against you.
* Build a REPL (read-eval-print loop) for TicTacToe to play interactively.

process.stdin.setEncoding('utf8');

const stdin = process.openStdin();

stdin.addListener('data', function(data) {

/\* do your tic tac toe interactive magic here \*/

});