Exercises

Just to give you some context, sometimes we make some not very smart mistakes (we are only human…). Tools like SonarLint are a valuable help in these situations. SonarLint statically analyzes code to quickly find issues.

For all the exercises proposed you should take some time to analyze the code WITHOUT using the tool. As an exceptionally talented programmer, you should be able to catch some of these issues without using it.

Then you can view the code on *VS Code* and *SonarLint* will outline the issues for you and give tips for fixing them! As easy as that!

You should take into account that, in some cases, fixing one issue reveals another set of brand new issues for you to enjoy! Fun right?

For additional information on the issues go here.

Exercise 1

file: *ex1.js*

solution: *sol1.js* (don't do it, any doubts ask us first!!)

The main goal of this exercise is for you to understand how the tool works and the different issues it identifies. In order to do that, four simple functions are presented to you. You should start fixing them in order.

Have fun 😃

Exercise 1.a.

In this function, it is expected that you fix 3 (+1 bonus) different issues.

```
// return the op between two POSITIVE numbers
// if the numbers are not >= 0, returns -1
// if op is not SUM, SUB, PROD or DIV, returns -2
function exA(a, b, op, op) {
  let res;
  if (a < 0 || b < 0)
    return - 1;
  switch (op) {
    case 'SUM':
      res = a + b;
      break;
    case 'SUB':
      res = a - b;
    case 'PROD':
      res = a * b;
      break;
```

```
    // sonarLint does not flag this, ESlint would...
    // lets pretend it does and fix it anyways :)
    case 'SUM':
      res = a + b;
      break;
    case 'DIV':
      res = a / b;
      break;
  }
  return res;
}
```

Exercise 1.b.

In this function, it is expected that you fix 4 different issues.

```
// returns INVALID OP if the result of the op between 2 nr is < 0
// returns ZERO if the result of the op between 2 nr is == 0
// returns POSITIVE if the result of the op between 2 nr is > 0
function exB(a, b, op) {
  const value = exA(a, b);
  if (value < 0) {
    return 'INVALID OP';
  } else if (value == 0) {
    return 'ZERO';
  } else if (value_ > 0) {
    return 'POSITIVE';
  } else if (value < 0) {
    return 'NEGATIVE';
  }
}
```

Exercise 1.c.

In this function, it is expected that you fix 5 different issues.

```
// returns true if the result of the op between two positive numbers is >=
0, false otherwise
// yes, there are better ways to do this, just do the exercise ;)
function exC(a, b, op) {
  const value = exB(a, b, op);
  if (vallue != 'INAVLID OP') {
    const c = true;
  } else {
    const b = false;
  }
```

```
  return c;
}
```

Exercise 1.d.

In this function, it is expected that you fix 2 different issues.

```
function exD(a, b) {
  const ops = ["SUB", "PROD", "DIV"];
  let i = 0;
  let max = exA(a,b, "SUM");
  while (i < 3) {
    const opRes = exA(a,b, ops[i])
    if (opRes > max)
      max = opRes
  }
  return max
}
```

Exercise 2

file: *ex2.js*

solution: *sol2.js* (nooo!)

This exercise presents a more “real” application of this tool. It is a very simple program that implements the tic tac toe game. As in the previous exercise, we advise you to fix the issues in order.

We are expecting you to encounter around 15/16 issues (it will depend on your resolution).

Have fuuuun! When you FINISH the exercise you can play the game in the index.html file 😉

```
const statusDisplay = document.querySelector('.game--status');

const gameActive = true;
let currentPlayer = 'X';
let gameState = ['', '', '', '', '', '', '', '', ''];

const winningMessage = () => `Player ${currentPlayer} has won!`;
const drawMessage = () => 'Game ended in a draw!';
const currentPlayerTurn = () => `It's ${currentPlayer}'s turn`;

statusDisplay.innerHTML = currentPlayerTurn();

const winningConditions = [
  [0, 1, 2],
  [3, 4, 5],
  [6, 7, 8],
  [0, 3, 6],
```

```
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];

  function handleCellPlayed(clickedCell, clickedCellIndex) {
    gameState[clickedCellIndex] = currentPlayer;
    clickedCell.innerHTML = currentPlayer;
  }

  function handlePlayerChange() {
    oldPlayer = currentPlayer;
    currentPlayer = currentPlayer === 'X' ? 'O' : 'X';
    statusDisplay.innerHTML = currentPlayerTurn();
  }

  function handleResultValidation() {
    let roundWon = false;
    //used for debugging only, could be removed :)
    const printRoundWon = !{};


    for (let i = 0; i <= 7; i += 1) {
      const winCondition = winningConditions[i];
      const a = gameState[winCondition[0]];
      const b = gameState[winCondition[1]];
      const c = gameState[winCondition[1]];
      c = gameState[winCondition[2]];

      if (a =='' || b == '' || c == '') {
        continue;
      }
      if (a == b && b == c) {
        roundWon = true;
        break;
      }
    }


    if (roundWon) {
      statusDisplay.innerHTML = winningMessage();
      gameActive = false;
      if(printRoundWon){
        console.log(roundWon)
      }
      return;
    }

    const roundDraw = !gameState.includes('');
    if (roundDraw) {
      statusDisplay.innerHTML = drawMessage();
      gameActive = false;
      return;
```

```
    }

    handlePlayerChange();
  }

  function handleCellClick(clickedCellEvent) {
    const clickedCell = clickedCellEvent.target;
    const clickedCellIndex = parseInt(clickedCell.getAttribute('data-cell-
  index'));

    if (gameState[clickedCellIndex] != '' || !gameActive) {
      return;
    }

    handleCellPlayed(clickedCell, clickedCellIndex);
    handleResultValidation();
  }

  function handleRestartGame() {
    gameActive = true;
    currentPlayer = 'X';
    gameState = ['', '', '', '', '', '', '', '', ''];
    statusDisplay.innerHTML = currentPlayerTurn();
    document.querySelectorAll('.cell').forEach((cell) => cell.innerHTML =
  '');
  }

  function setGameActive(value, oldValue) {
    gameActive = value;
  }

  document.querySelectorAll('.cell').forEach((cell) =>
  cell.addEventListener('click', handleCellClick));
  document.querySelector('.game--restart').addEventListener('click',
  handleRestartGame);
```

Christmas challenge (optional, but you should do it 😀)

(From Advent of code 2019)

It is Christmaaaaas time! No? Ok...

In the previous exercises you used the tool to fix issues of code already implemented. In this challenge we want you to experiment how this tool is used while you are developing code. It allows to fix issues as you write them! Profit.

Have fun and merry christmas? 🎄🎅

--- PART 01 ---

The Elves quickly load you into a spacecraft and prepare to launch.

At the first Go / No Go poll, every Elf is Go until the Fuel Counter-Upper. They haven't determined the amount of fuel required yet.

Fuel required to launch a given module is based on its mass. Specifically, to find the fuel required for a module, take its mass, divide by three, round down, and subtract 2.

For example:

- For a mass of 12, divide by 3 and round down to get 4, then subtract 2 to get 2.
- For a mass of 14, dividing by 3 and rounding down still yields 4, so the fuel required is also 2.
- For a mass of 1969, the fuel required is 654.
- For a mass of 100756, the fuel required is 33583.

The Fuel Counter-Upper needs to know the total fuel requirement. To find it, individually calculate the fuel needed for the mass of each module (*puzzle_input.txt*), then add together all the fuel values.

What is the sum of the fuel requirements for all of the modules on your spacecraft? (*puzzle_input.txt*)

--- PART 02 ---

During the second Go / No Go poll, the Elf in charge of the Rocket Equation Double-Checker stops the launch sequence. Apparently, you forgot to include additional fuel for the fuel you just added.

Fuel itself requires fuel just like a module - take its mass, divide by three, round down, and subtract 2. However, that fuel also requires fuel, and that fuel requires fuel, and so on. Any mass that would require negative fuel should instead be treated as if it requires zero fuel; the remaining mass, if any, is instead handled by wishing really hard, which has no mass and is outside the scope of this calculation.

So, for each module mass, calculate its fuel and add it to the total. Then, treat the fuel amount you just calculated as the input mass and repeat the process, continuing until a fuel requirement is zero or negative. For example:

- A module of mass 14 requires 2 fuel. This fuel requires no further fuel (2 divided by 3 and rounded down is 0, which would call for a negative fuel), so the total fuel required is still just 2.
- At first, a module of mass 1969 requires 654 fuel. Then, this fuel requires 216 more fuel (654 / 3 - 2). 216 then requires 70 more fuel, which requires 21 fuel, which requires 5 fuel, which requires no further fuel. So, the total fuel required for a module of mass 1969 is 654 + 216 + 70 + 21 + 5 = 966.
- The fuel required by a module of mass 100756 and its fuel is: 33583 + 11192 + 3728 + 1240 + 411 + 135 + 43 + 12 + 2 = 50346.

What is the sum of the fuel requirements for all of the modules on your spacecraft when also taking into account the mass of the added fuel? (Calculate the fuel requirements for each module separately, then add them all up at the end.)

Solution:

- Part 01: 3342946
- Part 02: 5011553