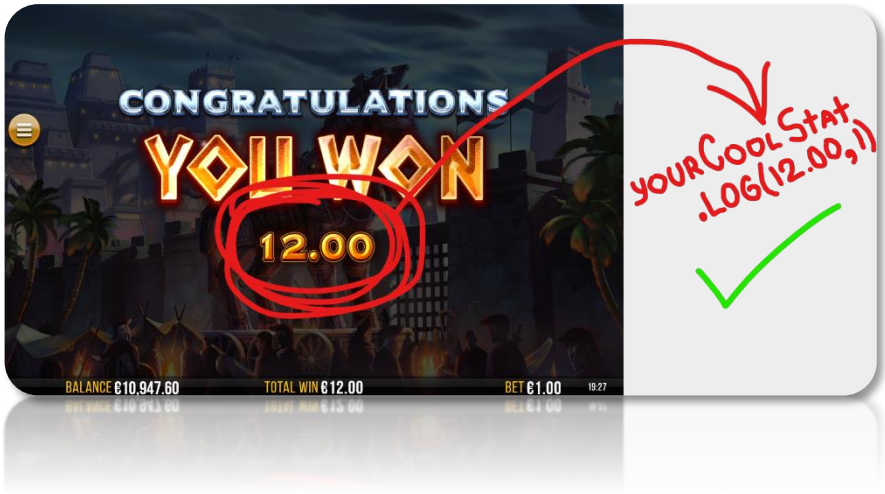
4TP: TypeScript basic test task 3

Description

Create abstraction that will allow storing wins statistics as *fast* as *possible*. All optimizations are welcome!



Part #1/3: initialize project.

1. Create new empty node.js project, "`npm int`", etc.
2. Install locally TypeScript with NPM (save package as developer dependency).
3. Initialize TypeScript with default config:
    a. The source directory with all code is "`./src/scripts`" in the root of the project.
    b. The result – compiled JS directory: "`./dist/js`".
    c. Any additional configuration is optional, not needed. The default config file is just fine.
4. Create "`index.ts`" file in the "`./src/scripts`" (it will be compiled to index.js on the TS compilation). It will be an entry point of test app.

Part #2/3: create statistic abstraction logic with TypeScript

Note: each class and interface should be in the separate file in the "`./src/scripts`" with structure by your choice.

Tasks:

1. Create an class that will store wins statistics with 4 methods:
    a. "`log(winAmount: number, hitCount: number): void {…}`" – the method saves new wins into the statistics.
        i. The first argument is the win amount, which can be an integer or float number. If the number is float it should be rounded to the nearest with a precision of 1 digit after the dot. Few examples:
            1. `0.3612` -> `0.4`
            2. `0.434` -> `0.4`
            3. `2` -> `2`

4. `0.1999999` -> `0.2`
5. `0.5` -> `0.5`
6. `1.000000001` -> `1`
7. `0.999999999` -> `1`

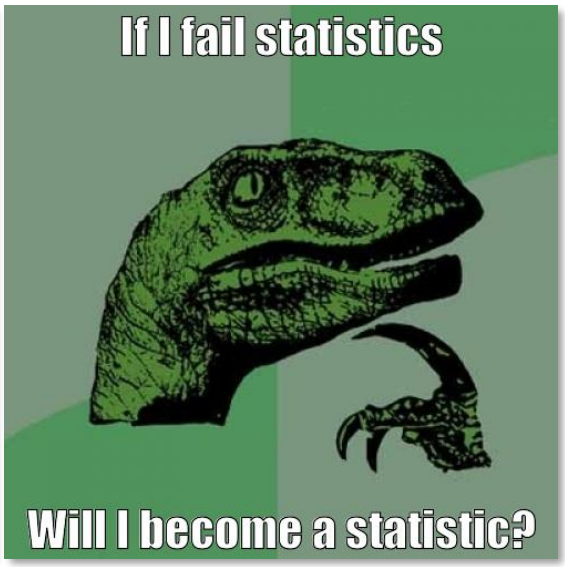
ii. A second argument is the number of such wins that happen at once.
iii. All arguments should be validated! The complete validation of the input arguments should be made: `winAmount` can be only 0 or any positive, finite number. The `hitCount` is any positive finite integer number.
iv. The complexity of this operation should be **O(1)**.

b. "`getHitCount(winAmount: number): number {…}`" – returns the hit count for specified win amount.
c. "`merge(anotherStat: %YourType%): void {…}`" – merges another statistic object into `this` instance, e.g. if `this` stat has 5 hits of zero wins, and another stat 80, then after merge `this` should contain 85 hits of zeros.
d. "`print(): void {…}`" – prints all collected data into the console. See attachment #1 for the format information.

2. Copy attachment #2 into the project and run the simulation with your cool statistic in the "`index.ts`" with "`Simulation.runSimulation(…)`".

Part #3/3: test & publish.

1. Make sure (or compile) TypeSctipt to JS in the "`./dist/js`" directory.
2. Configurate the "`package.json`" file to be able to start our project with the "start" command, e.g.: "`npm run start`" will start the "`./dist/js/index.js`" with node.js.
3. Test project to be able to start it from console, like:
   a. "`npm run start`".
4. Publish project to the GitHub or any other git service of your choice.
5. Send link of the published repository.



Feel free to contact in case of comments or questions.

Good luck!

Attachment #1/2: the example of the “print()” method output.

```
Total win amount: %YOUR_SUM_OF_ALL_WINS%.
The average win amount: %YOUR_AVERAGE_WIN_AMOUNT%.
The smallest non-zero win is %YOUR_NUMBER%, the biggest is %YOUR_NUMBER%.

All unique wins (sorted 0…9):
1. %WIN_AMOUNT%: %HIT_COUNT%
2. %WIN_AMOUNT%: %HIT_COUNT%
3. %WIN_AMOUNT%: %HIT_COUNT%
4. %WIN_AMOUNT%: %HIT_COUNT%
5. %WIN_AMOUNT%: %HIT_COUNT%
6. %WIN_AMOUNT%: %HIT_COUNT%
…and all other.
```

or:

```
Total win amount: 1546.15.
The average win amount: 2.456.
The smallest non-zero win is 0.1, the biggest is 4.2.

All unique wins (sorted 0…9):
1. 0: 54655
2. 0.1: 54
3. 0.7: 122
4. 0.9: 591
5. 2.6: 97
6. 3.1: 314
…and all other.
```

Attachment #2/2: the file with code of the simulation that should be used to test the create wins statistics.

```
interface LoggableStat {
    log(winAmount: number, hitCount: number): void;
}
interface MergeableStat {
    merge(anotherStat: MergeableStat): void;
}
interface TestableStat {
    getHitCount(winAmount: number): number;
}
export interface Stat extends LoggableStat, MergeableStat, TestableStat {
    print(): void;
}

export type CreateStatFn = () => Stat;

export class Simulation {
    private static readonly logIterationCount = 50000;
    private static readonly statsToTestCount = 10;

    static runSimulation(createStatFn: CreateStatFn): number {
        if (createStatFn == null)
            throw Error('create new stat function not specified');

        const startTime = Date.now();

        const resStat = Simulation.runSingleSim(createStatFn);
        resStat.print();

        const durationMs = Date.now() - startTime;
        console.log(`Simulation took ${durationMs}ms.`);

        return durationMs;
    }

    private static runSingleSim(createStatFn: CreateStatFn): Stat {
        // 1/4. Create statistics.
        const stats = [];
        for (let i = 0; i < Simulation.statsToTestCount; i++) {
            stats.push(createStatFn());
        }

        // 2/4. Fill.
        for (const stat of stats) {
            Simulation.fillStat(stat);
        }

        // 3/4. Merge all together.
        const resStat = Simulation.mergeStats(createStatFn, stats);

        // 4/4. Test.
        Simulation.testFinalStatWins(resStat);

        return resStat;
    }

    private static fillStat(stat: LoggableStat): void {
```

```
        for (let i = 0; i < Simulation.logIterationCount; i++) {
            if (Math.random() < 0.5) {
                // 50% chance of no win.
                stat.log(0, 1);
                continue;
            }

            const rndWinAmount =
                Math.random() + Math.floor(Math.random() * 3) + 1; // [1,3.99999999999999
99] -> [1,4] after rounding.
            const rndHitCount = Math.floor(Math.random() * 2) + 1; // [1,2]
            stat.log(rndWinAmount, rndHitCount);
        }
    }

    private static mergeStats(
        createStatFn: CreateStatFn,
        stats: MergeableStat[]
    ): Stat {
        const resStat = createStatFn();
        for (const stat of stats) {
            resStat.merge(stat);
        }

        return resStat;
    }

    private static testFinalStatWins(stat: TestableStat): void {
        const expectedZerosHitCount =
            (Simulation.logIterationCount / 2) * // 50% of getting zero in each test.
            Simulation.statsToTestCount;
        Simulation.testSingleWin(stat, 0, expectedZerosHitCount);

        const winAmountSmallestIncl = 1;
        const smallestWinAmountStep = 0.1;
        const winAmountBiggestIncl = 4;

        // Test all values in the middle.
        const expectedMiddleHitCount = expectedZerosHitCount / 2 / 10;
        for (
            let winAmount = winAmountSmallestIncl + smallestWinAmountStep;
            winAmount < winAmountBiggestIncl;
            winAmount += smallestWinAmountStep
        ) {
            Simulation.testSingleWin(
                stat,
                winAmount,
                expectedMiddleHitCount
            );
        }

        // Test lower and upper range.
        const expectedEdgesZerosHitCount = expectedMiddleHitCount / 2;
        Simulation.testSingleWin(
            stat,
            winAmountSmallestIncl,
            expectedEdgesZerosHitCount
        );
        Simulation.testSingleWin(
            stat,
            winAmountBiggestIncl,
```

```
            expectedEdgesZerosHitCount
        );

        // Test out of the range.
        Simulation.testSingleWin(
            stat,
            winAmountSmallestIncl - smallestWinAmountStep,
            0
        );
        Simulation.testSingleWin(
            stat,
            winAmountBiggestIncl + smallestWinAmountStep,
            0
        );
    }

    private static testSingleWin(
        stat: TestableStat,
        testableWinAmount: number,
        expectedHitsCount: number
    ): void {
        const discrepancy = 0.1;
        const currWinHitCount = stat.getHitCount(testableWinAmount);
        if (
            !Simulation.checkIsValueInRange(
                currWinHitCount,
                expectedHitsCount,
                discrepancy
            )
        ) {
            throw new Error(
                `Statistics contains incorrect data: there are ${currWinHitCount} hits of
 "${testableWinAmount}" but expected ${expectedHitsCount} +/-${
                    discrepancy * 100
                }% hits.`
            );
        }
    }

    private static checkIsValueInRange(
        currentValue: number,
        referenceValue: number,
        discrepancy: number
    ): boolean {
        return (
            currentValue === referenceValue ||
            (currentValue > referenceValue * (1 - discrepancy) &&
                referenceValue * (1 + discrepancy) > currentValue)
        );
    }
}
```