Junioraufgabe-1: Bällebad

Team-ID: 00596

Name: Sophie-RS

23. Oktober 2025

Programmiersprache: TypeScript (NodeJS v22.16.0)

Inhaltsverzeichnis

Problembeschreibung

Die Alan-Turing-Schule möchte neue Medizinbälle anschaffen. Dafür wird ein Programm benötigt, das überprüft, wie viele Bälle maximal gebraucht werden. Das Programm soll anhand der Stundenpläne berechnen, zur welcher Zeit die meisten Bälle benötigt werden und diese Anzahl ausgeben.

Lösungsidee

Um das beschriebene Problem zu lösen, wird zunächst untersucht, welche Klassen zur gleichen Zeit Unterricht haben und zwar unter Berücksichtigung von Uhrzeit und Wochentag.
Dazu wird überprüft, wann jede Klasse mit dem Unterricht beginnt und wann sie endet. Anschließend werden diese Start- und Endzeiten klassenübergreifend miteinander verglichen.
Wenn zwei Klassen dieselbe Start- und Endzeit haben, bedeutet das, dass sie gleichzeitig Unterricht haben. Trifft das nicht zu, wird überprüft, ob die

Startzeit der ersten Klasse nach der Startzeit der zweiten Klasse liegt, aber noch vor deren Endzeit. In diesem Fall überschneiden sich die Unterrichtszeiten, und die Klassen haben zur selben Zeit Unterricht.
Alle Klassen, die zur gleichen Zeit Unterricht haben, werden zu einer gemeinsamen Kombination zusammengefasst. Hat eine Klasse alleine Unterricht, wird sie ebenfalls in eine eigene Kombination aufgenommen, die dann nur aus dieser einen Klasse besteht.

Nachdem alle Kombinationen gebildet wurden, wird berechnet, wie viele Schüler jeweils in einer Kombination sind. Anschließend wird überprüft, ob die Klassen innerhalb einer Kombination tatsächlich gleichzeitig Unterricht haben. In der ersten Auswahl wurde nämlich nur geprüft, welche Klassen überhaupt Überschneidungen haben nicht, ob alle Klassen einer Kombination sich gegenseitig überschneiden.
Dazu werden erneut die Start- und Endzeiten verglichen. Nachdem diese Prüfung abgeschlossen ist, kann erneut berechnet werden, wie viele Schüler in der jeweiligen Kombination sind. Diese Anzahl bestimmt schließlich, wie viele Bälle für diese Kombination benötigt werden.

Zum Schluss betrachten wir die Kombination mit den meisten Schülern also diejenige, die die meisten Bälle benötigt. Anschließend wird ermittelt, zu welcher Uhrzeit und an welchem Wochentag diese Klassen Unterricht haben. Dieses Ergebnis stellt schließlich unsere Lösung dar.

Umsetzung

Wir haben das gesamte Programm in TypeScript geschrieben und mit NodeJS, Version 22.16.0, ausgeführt. Sowohl der TypeScript- als auch der JavaScript-Code sind in der ZIP-Datei enthalten. In dieser Dokumentation werden wir jedoch nur den Code aus der TypeScript-Version erläutern, da die JavaScript-Version auf dem TypeScript-Code basiert und daraus kompiliert wurde.

Wir haben uns dazu entschieden, das Programm konsolenbasiert zu programmieren. Das bedeutet, dass jegliche Ein- und Ausgabe innerhalb der Konsole erfolgt.
Um der Konsolenausgabe einen schöneren Stil zu geben, haben wir das Modul *consoleLogger.ts* in das Programm importiert. Dabei handelt es sich um eine Sammlung selbst geschriebener Funktionen, die das farbige Schreiben in der Konsole mithilfe von Color-Codes vereinfachen.
Sie funktionieren so, dass man bestimmte Parameter an eine Funktion

übergeben kann, aus denen anschließend ein String erstellt wird, der die entsprechenden Color-Codes enthält.
Da dies nichts mehr mit der eigentlichen Aufgabe zu tun hat, werden wir hier weder auf die genaue Funktionsweise noch auf die konkrete Nutzung in diesem Programm eingehen.
Wenn wir in dieser Dokumentation von „in der Konsole ausgeben“ schreiben, ist damit ein Text gemeint, der von einer dieser Funktionen erzeugt wurde.

Um die Konsoleneingabe zu realisieren, haben wir das npm-Package *readline* in das Programm importiert. Von diesem Package nutzen wir die Funktion *createInterface()*, die ein Objekt zurückgibt, über das die Methode *question(question: string, callback: Function)* aufgerufen werden kann, um den Benutzer nach einer Eingabe zu fragen.

Das Programm beginnt damit, den Nutzer nach dem Namen der Datei zu fragen. Wird ein Name eingegeben, der keiner vorhandenen Datei entspricht, gibt das Programm eine Fehlermeldung aus und weist darauf hin, die Schreibweise zu überprüfen. Die angegebene Datei muss sich außerdem im Test-Ordner befinden.
Existiert die angegebene Datei, verwendet das Programm die Funktion *textToString(file: string)*. Diese nutzt das *fs-Modul*, um die angegebene Textdatei mit *fs.readFileSync(filepath: string, encoding: string)* in einen nutzbaren String umzuwandeln.

```
function textToString(file: string) {
  let content = "";
  try {
    content = fs.readFileSync(file, "utf-8");
  } catch (err) {
    content = "error";
  }
  return content;
}

function main(path: string) {
  const content = textToString(path);

  if (content === "error") {
    console.log(fehlerNachricht);
    return;
  }
```

Nachdem die Datei in einen String umgewandelt wurde, überprüft das Programm zunächst, welche Klassen gleichzeitig Unterricht haben. Dafür wird die Funktion *checkTogetherness(text: string)* verwendet.
Diese Funktion teilt den Text zuerst in Absätze auf, indem sie die Methode *split(...)* nutzt. Dadurch werden die Informationen der einzelnen Klassen getrennt. Anschließend iteriert das Programm mit einem for-Loop über dieses Array. Innerhalb dieser Schleife wird jedes Element erneut nach Leerzeichen aufgeteilt, um Zugriff auf alle Informationen einer Klasse zu erhalten.
Das Programm entnimmt daraus die wichtigsten Daten, also den Wochentag sowie die Start- und Endzeit. Innerhalb des ersten for-Loops wird dann ein

```
// Teste jede Klasse mit jeder anderen Klasse
for (const l of lines) {
  if (l === lines[0]) {
    continue;
  }

  const newElements = l.split(" ");
  const day = newElements[1];
  const newStartTime: number = Number(newElements[2]);
  const newEndTime: number = Number(newElements[3]);

  // Überprüft ob die Startzeit grösser oder gleich die neue
  // Startzeit ist und ob die Neuestartzeit kleiner als die end Zeit ist.
  if (day === startDay) {
    if (
      (startTime <= newStartTime && newStartTime < endTime) ||
      (startTime === newStartTime && endTime === newEndTime)
    ) {
      if (result === "") {
        result += turn.toString();
      } else {
        result += "," + turn.toString();
      }
    }
  }

  turn += 1;
}
```

zweiter for-Loop gestartet, der wiederum alle Klassen durchläuft. Auch hier werden die relevanten Informationen in Variablen gespeichert.
Nun hat das Programm zwei Start- und Endzeiten, die es miteinander vergleicht. Zuerst wird überprüft, ob die Tage übereinstimmen. Falls nicht, fährt das Programm direkt mit dem nächsten Element im Array fort. Stimmen die Tage jedoch überein, werden die Start- und Endzeiten der beiden Klassen verglichen.
Wenn sowohl Start- als auch Endzeit identisch sind, wird die Klasse zu einem String hinzugefügt und durch ein Komma getrennt. Trifft das nicht zu, prüft das Programm, ob die Startzeit der ersten Klasse später als die der zweiten liegt, aber noch vor deren Endzeit. Ist das der Fall, wird die Klasse ebenfalls in den String aufgenommen und mit einem Komma getrennt.
Am Ende des inneren for-Loops wird dieser String in ein Array eingefügt, das alle Klassenkombinationen enthält.

Nachdem die Klassenkombinationen ermittelt wurden, berechnet das Programm, welche Kombination die meisten Schüler hat. Diese Aufgabe übernimmt die Funktion *lookForStrongest()*.
In dieser Funktion wird mithilfe eines for-Loops überprüft, wie viele Schüler jede Kombination enthält. Anschließend wird kontrolliert, ob die aktuelle Kombination mehr Schüler hat als die bisher gespeicherte. Falls noch keine Kombination gespeichert ist, wird die aktuell geprüfte Kombination gespeichert.

```
if (strongest < content) {
  // Startzeit berechnen
  lowestEndTime = 25;
  lowestStartTime = -1;

  // Vergleiche die Zeitspanne
  for (const num of newNumbers) {
    if (lowestStartTime <= startTimes[num]) {
      lowestStartTime = startTimes[num];
    }

    if (lowestEndTime > endTimes[num]) {
      lowestEndTime = endTimes[num];
    }
  }
  // Vergleiche den Tag.
  day = dayList[newNumbers[0]];
  strongestClassTurn = turn;
  strongest = content;
} else {
  notStrongest.push(turn);
}
```

Zusätzlich berechnet die Funktion den Zeitraum des Unterrichts. Wenn die aktuelle Kombination größer ist als die gespeicherte, werden in einem weiteren for-Loop die Start- und Endzeiten der jeweiligen Kombination überprüft. Liegt die Startzeit später als die bisher gespeicherte, wird sie aktualisiert, liegt die Endzeit früher, wird auch diese angepasst. Falls noch keine Zeiten gespeichert sind, werden die aktuellen Werte übernommen. Außerdem wird der Wochentag der Kombination ausgelesen und in einer globalen Variable gespeichert. Zum Schluss gibt die Funktion die Schüleranzahl zurück.

Nachdem das Programm die Kombination mit den meisten Schülern gefunden hat, wird durch die Funktion *checkIfCombinationsMatch()* geprüft, ob diese Kombination tatsächlich gültig ist. Diese Prüfung ist notwendig, da in *checkTogetherness()* nur festgestellt wird, welche Klassen Überschneidungen haben jedoch nicht, ob alle Klassen innerhalb einer Kombination auch untereinander Unterricht haben. Genau dieses Problem behebt *checkIfCombinationsMatch().*

Die Funktion arbeitet ähnlich wie *checkTogetherness()*: Sie extrahiert zunächst die wichtigsten Daten der Klassen (Tag, Start- und Endzeit) und vergleicht dann alle Klassenpaare miteinander ebenfalls durch zwei verschachtelte for-Loops. Wenn festgestellt wird, dass zwei Klassen nicht gleichzeitig Unterricht haben, wird dieses Klassenpaar als Kombination in ein Array namens *missingElements* aufgenommen.

```
for (const c of realClasses) {
  // Jede Klasse mit jeder anderen Klasse prüfen,
  // ob sie wirklich zusammen Unterricht haben.

  const testClass: string = lines[Number(c) + 1];
  const baseClass = testClass.split(" ");
  const compStartTime: number = Number(baseClass[2]);
  const compEndTime: number = Number(baseClass[3]);

  for (const newClass of realClasses) {
    const newTestClass = lines[Number(newClass) + 1];
    const newBaseClass = newTestClass.split(" ");
    const newStart: number = Number(newBaseClass[2]);
    const newEnd: number = Number(newBaseClass[3]);

    if (
      (compStartTime <= newStart && newStart < newEnd) ||
      (compStartTime === newStart && compEndTime === newEnd) ||
      (newStart <= compStartTime && newEnd > compStartTime)
    ) {
    } else {
      missingElements.push(newClass.toString() + "," + c.toString());
    }
  }
}
```

Nachdem beide Schleifen durchlaufen wurden, überprüft das Programm, welche Klassen aus der Kombination entfernt werden müssen. Ist das Array *missingElements* leer, bleibt die Kombination unverändert. Befinden sich jedoch Einträge darin, wird zunächst geprüft, ob eine der beiden Klassen bereits in der *deadList* steht einer Liste, die alle gelöschten Klassen enthält.
Wenn keine der beiden Klassen in dieser Liste steht, prüft das Programm, welche der beiden mehr Konflikte verursacht. Steht zum Beispiel eine Klasse mit zehn anderen im Konflikt, wird bevorzugt diese Klasse gelöscht. Haben beide Klassen gleich viele Konflikte, entscheidet die Schüleranzahl: Die Klasse

```
if (missingElements.length === 0) {
  newArray.push(classes[index]);
} else {
  for (const pair of missingElements) {
    const numbers = pair.split(",");
    const indexNums = numbers.map(Number);
    const firstNum: number = Number(lines[indexNums[0] + 1].split(" ")[4]);
    const secondNum: number = Number(lines[indexNums[1] + 1].split(" ")[4]);

    // Überprüfen ob es schon gelöscht wurde
    if (deadList.includes(indexNums[0]) || deadList.includes(indexNums[1])) {
      continue;
    }

    const first = getInstances(missingElements.toString().split(",").map(Number), indexNums[0]);
    const second = getInstances(missingElements.toString().split(",").map(Number), indexNums[1]);

    if (first > second) {
      deadList.push(indexNums[0]);
    } else if (first < second) {
      deadList.push(indexNums[1]);
    } else {
      if (firstNum > secondNum) {
        deadList.push(indexNums[1]);
      } else {
        deadList.push(indexNums[0]);
      }
    }
  }
}
```

mit weniger Schülern wird auf die *deadList* gesetzt. Anschließend wird die ursprüngliche Kombination bereinigt, indem alle Klassen, die sich in der *deadList* befinden, daraus entfernt werden. So entsteht die endgültige Kombination. Danach wird erneut überprüft, welche Kombination nun die größte ist, da die bereinigte Kombination eventuell weniger Schüler enthält. Sollte das der Fall sein, wird derselbe Prozess bis zu fünfmal wiederholt.

Zum Schluss gibt das Programm das Ergebnis in der Konsole aus inklusive Startzeit, Endzeit und Wochentag. Falls Start- und Endzeit identisch sind, wird nur die Uhrzeit ohne „von–bis“-Angabe ausgegeben.

Werkzeuge

- **Visual Studio Code:** Wir haben das gesamte Programm mit der IDE Visual Studio Code geschrieben. Diese Umgebung bietet Code-Completion und eine integrierte TypeScript-Unterstützung, durch die die automatische Codevervollständigung ermöglicht wird. Jeglicher Code, der in dieser Dokumentation dargestellt ist, wurde in Visual Studio Code fotografiert.

- **NPM:** NPM (Node Package Manager) ist ein Paketmanager, mit dem das Programm zusätzliche Packages nutzen kann. In diesem Projekt sind das Readline-Package und die TypeScript-Declarations für das fs-Modul von JavaScript installiert. NPM vereinfacht außerdem die Ausführung des Programms, da damit Skripte erstellt werden können, die mehrere Befehle nacheinander ausführen.

- **ChatGPT (GPT-5):** ChatGPT wurde hauptsächlich zur Korrektur von Rechtschreibung und Satzbau verwendet, wobei wir versucht haben, unseren eigenen Schreibstil so weit wie möglich beizubehalten

- **Internet:** Das Internet wurde genutzt, um bestimmte Funktionen in Javascript zu recherchieren.
- **Apple Pages:** Apple Pages wurde zum Schreiben dieser Dokumentation verwendet. Dabei wurde keine integrierte KI-Funktion eingesetzt.
- **Google-Docs:** Google docs wurde benutzt, um zusammen an der Dokumentation zu arbeiten.
- **Familie:** Diese Dokumentation haben Teile unserer Familien gelesen, um Satzbau und Aufgaben korrekt zu überprüfen.

Beispiele

Beispiel00:

```
-Jugendwettbewerb-2025-43-Runde-3- Junioraufgabe-1 Sophie-RS
Bitte nenne mir den name deiner Datei. (Sie muss in dem Test-Ordner sein!, das '.txt' bitte auslassen.): ball00

--Ergebnis----------
Es werden 60 Bälle gebraucht, und zwar am Montag, von 15 bis 16 Uhr!
--------------------
```

Beispiel01:

```
-Jugendwettbewerb-2025-43-Runde-3- Junioraufgabe-1 Sophie-RS
Bitte nenne mir den name deiner Datei. (Sie muss in dem Test-Ordner sein!, das '.txt' bitte auslassen.): ball01

--Ergebnis----------
Es werden 79 Bälle gebraucht, und zwar am Montag, von 10 bis 11 Uhr!
--------------------
```

Beispiel02:

```
-Jugendwettbewerb-2025-43-Runde-3- Junioraufgabe-1 Sophie-RS
Bitte nenne mir den name deiner Datei. (Sie muss in dem Test-Ordner sein!, das '.txt' bitte auslassen.): ball02

--Ergebnis----------
Es werden 157 Bälle gebraucht, und zwar am Montag, von 9 bis 10 Uhr!
--------------------
```

Beispiel03:

```
-Jugendwettbewerb-2025-43-Runde-3- Junioraufgabe-1 Sophie-RS
Bitte nenne mir den name deiner Datei. (Sie muss in dem Test-Ordner sein!, das '.txt' bitte auslassen.): ball03

--Ergebnis----------
Es werden 135 Bälle gebraucht, und zwar am Mittwoch, von 14 bis 15 Uhr!
--------------------
```

Beispiel04:

```
-Jugendwettbewerb-2025-43-Runde-3- Junioraufgabe-1 Sophie-RS
Bitte nenne mir den name deiner Datei. (Sie muss in dem Test-Ordner sein!, das '.txt' bitte auslassen.): ball04

--Ergebnis----------
Es werden 31 Bälle gebraucht, und zwar am Montag, von 10 bis 12 Uhr!
--------------------
```

Beispiel05:

```
-Jugendwettbewerb-2025-43-Runde-3- Junioraufgabe-1 Sophie-RS
Bitte nenne mir den name deiner Datei. (Sie muss in dem Test-Ordner sein!, das '.txt' bitte auslassen.): ball05

--Ergebnis----------
Es werden 60 Bälle gebraucht, und zwar am Donnerstag, von 8 bis 10 Uhr!
--------------------
```

Beispiel06:

```
-Jugendwettbewerb-2025-43-Runde-3- Junioraufgabe-1 Sophie-RS
Bitte nenne mir den name deiner Datei. (Sie muss in dem Test-Ordner sein!, das '.txt' bitte auslassen.): ball06

--Ergebnis----------
Es werden 90 Bälle gebraucht, und zwar am Dienstag, von 8 bis 15 Uhr!
--------------------
```

Beispiel07:

```
-Jugendwettbewerb-2025-43-Runde-3- Junioraufgabe-1 Sophie-RS
Bitte nenne mir den name deiner Datei. (Sie muss in dem Test-Ordner sein!, das '.txt' bitte auslassen.): ball07

--Ergebnis----------
Es werden 135 Bälle gebraucht, und zwar am Mittwoch, von 14 bis 15 Uhr!
--------------------
```

Eigene Beispiele

Eigener-test:

```
-Jugendwettbewerb-2025-43-Runde-3- Junioraufgabe-1 Sophie-RS
Bitte nenne mir den name deiner Datei. (Sie muss in dem Test-Ordner sein!, das '.txt' bitte auslassen.): eigenertest

--Ergebnis----------
Es werden 41 Bälle gebraucht, und zwar am Freitag, von 9 bis 13 Uhr!
--------------------
```

Eigener-test1:

```
-Jugendwettbewerb-2025-43-Runde-3- Junioraufgabe-1 Sophie-RS
Bitte nenne mir den name deiner Datei. (Sie muss in dem Test-Ordner sein!, das '.txt' bitte auslassen.): eigenertest1


--Ergebnis----------
Es werden 40 Bälle gebraucht, und zwar am Mittwoch, von 10 bis 20 Uhr!
--------------------
```

Quellcode

Die Main Funktion. Sie ruft die wichtigen Funktion zum Berechnen des Ergebnisses auf

```
function main(path: string) {
  const content = textToString(path);

  if (content === "error") {
    console.log(fehlerNachricht);
    return;
  }

  // Überprüft welche Klassen zusammen sind.
  const classes = checkTogetherness(content);

  // Überprüft welche Kombination die Meisten Schüler hat.
  const midResult = lookForStrongest(classes);

  const midCheck = checkIfCombinationsMatch(content, classes, strongestClassTurn);

  const result = lookForStrongest(midCheck);

  let realMidCheck = midCheck;
  let realResult = 0;
  let realClasses = classes

  // nochmal prüfen, 5 mal.
  // (braucht man in den Beispielen eigentlich nicht, jedoch könnte es in der Theorie passieren.)
  for (let i = 0; i < 5; i++) {
    realMidCheck = checkIfCombinationsMatch(content, realClasses, strongestClassTurn);
    realClasses = realMidCheck
    realResult = lookForStrongest(realMidCheck);
  }

  console.log(headerErgebnis);

  // Das Fertige Ergebnis!
  if (lowestEndTime != lowestStartTime) {
    console.log(ergebnisZweiZeiten(result, day, lowestStartTime, lowestEndTime));
  } else {
    console.log(ergebnisEineZeit(result, day, lowestEndTime));
  }

  console.log(endErgebnis);
}
```

Überprüft welche Klassen zusammen Unterricht haben

```
function checkTogetherness(text: string) {
  const lines = text.replace(/\r/g, "").split("\n");

  const classes: string[] = [];

  for (const line of lines) {

    // Ignoriere die erste Line (diese gibt nur die Anzahl der Klassen an)
    if (line === lines[0]) {
      continue;
    }

    const elements = line.split(" ");

    const startDay = elements[1];
    const startTime: number = Number(elements[2]);
    const endTime: number = Number(elements[3]);

    classCount.push(elements.map(Number)[4]);
    dayList.push(startDay);
    startTimes.push(elements.map(Number)[2]);
    endTimes.push(elements.map(Number)[3]);

    let turn = 0;
    let result = "";

    // Teste jede Klasse mit jeder anderen Klasse
    for (const l of lines) {
      if (l === lines[0]) {
        continue;
      }

      const newElements = l.split(" ");
      const day = newElements[1];
      const newStartTime: number = Number(newElements[2]);
      const newEndTime: number = Number(newElements[3]);

      // Überprüft ob die Startzeit grösser oder gleich die neue
      // Startzeit ist und ob die Neuestartzeit kleiner als die end Zeit ist.
      if (day === startDay) {
        if (
          (startTime <= newStartTime && newStartTime < endTime) ||
          (startTime === newStartTime && endTime === newEndTime)
        ) {
          if (result === "") {
            result += turn.toString();
          } else {
            result += "," + turn.toString();
          }
        }
      }

      turn += 1;
    }

    // Sortieren durch Komma.
    classes.push(result);
  }

  classes.splice(-1);
  return classes;
}
```

Überprüft ob die Kombinationen untereinander zusammen Unterricht haben

```
function checkIfCombinationsMatch(text: string, classes: string[], index: number) {

  const realClasses = classes[index].split(",");
  const missingElements: string[] = [];
  const lines = text.replace(/\r/g, "").split("\n");

  for (const c of realClasses) {
    // Jede Klasse mit jeder anderen Klasse prüfen,
    // ob sie wirklich zusammen Unterricht haben.

    const testClass: string = lines[Number(c) + 1];
    const baseClass = testClass.split(" ");
    const compStartTime: number = Number(baseClass[2]);
    const compEndTime: number = Number(baseClass[3]);

    for (const newClass of realClasses) {
      const newTestClass = lines[Number(newClass) + 1];
      const newBaseClass = newTestClass.split(" ");
      const newStart: number = Number(newBaseClass[2]);
      const newEnd: number = Number(newBaseClass[3]);

      if (
        (compStartTime <= newStart && newStart < newEnd) ||
        (compStartTime === newStart && compEndTime === newEnd) ||
        (newStart <= compStartTime && newEnd > compStartTime)
      ) {
      } else {
        missingElements.push(newClass.toString() + "," + c.toString());
      }
    }
  }

  const newArray: string[] = [];
  const deadList: number[] = [];

  if (missingElements.length === 0) {
    newArray.push(classes[index]);
  } else {
    for (const pair of missingElements) {
      const numbers = pair.split(",");
      const indexNums = numbers.map(Number);
      const firstNum: number = Number(lines[indexNums[0] + 1].split(" ")[4]);
      const secondNum: number = Number(lines[indexNums[1] + 1].split(" ")[4]);

      // Überprüfen ob es schon gelöscht wurde
      if (deadList.includes(indexNums[0]) || deadList.includes(indexNums[1])) {
        continue;
      }

      const first = getInstances(missingElements.toString().split(",").map(Number), indexNums[0]);
      const second = getInstances(missingElements.toString().split(",").map(Number), indexNums[1]);

      if (first > second) {
        deadList.push(indexNums[0]);
      } else if (first < second) {
        deadList.push(indexNums[1]);
      } else {
        if (firstNum > secondNum) {
          deadList.push(indexNums[1]);
        } else {
          deadList.push(indexNums[0]);
        }
      }
    }
  }
```

```
  const cleanDeadList: number[] = [...new Set(deadList)];
  let finalString: string = "";

  if (missingElements.length != 0) {
    for (const c of realClasses) {
      let keep = true;

      for (const member of cleanDeadList) {
        if (c === member.toString()) {
          keep = false;
        }
      }

      if (keep) {
        if (finalString === "") {
          finalString += c;
        } else {
          finalString += "," + c;
        }
      }
    }

    newArray.push(finalString);
  }

  const weakArray = notStrongest.toString().split(",");
  newArray.push(...weakArray);

  return newArray;
}
```