The Logical Imitation Game

For the next sequence of tasks, we'll be implementing, in stages, a version of "Turing Machine". As with the previous sequence (Parts 1 & 2), we'll start in Java to explore the structural components, then move to Python for the interaction components.

This task will also step us a little closer to realistic software development (though still at a fair distance - we have a clear idea of what we're doing for a start).

The Starting Point

The scaffold contains a Runner class, and nothing else. `Runner` is to facilitate you running any test/debugging code you need to complete the task, however it is not part of the tests.

You will have to create 6 `public` classes, the `Machine` class, `IncorrectDataException`, the `interface Verifier` and its four implementing classes `Verifier03`, `Verifier08`, `Verifier11`, and `Verifier21`.

Unlike with Part 1, you will need code that *compiles* before the tests will run - this means that you will not only have to create the appropriate classes, you will need to implement *method stubs* for all the required methods (a method stub is just the correct method header and a functioning `return` statement if

necessary, it doesn't have other code [let alone working code]). One way to do this is to practice reading the error messages...

The `IncorrectDataException`

We'll cover the exception first because it's simple. You do not need to implement any of the complicated functions of exceptions, and can rely on inheritance to do the work needed for this task. This of course means that when you create it, it must inherit from the correct parent class. The obvious choice is the correct one here.

The `Verifier` interface and its implementations

The `Verifier` interface should have the following `public` method:

- `check` that takes two `int` arrays (guess and answer, in that order) and returns `true` if the guess matches the criteria specified.

Each class that implements `Verifier` should also override the `toString` method to return the description of the verifier.

For the task, you will only need to implement 4 of the 48 verifiers. The test cases *may* include other verifiers. The four implementing classes and their logical functionality are:

- `Verifier03` - "Compares the 2nd digit to the value '3' (is it < 3, = 3 or > 3?)".

    - For example, if the answer is `[2, 4, 1]` then a guess of `[1, 2, 3]` should return `false` because `4 > 3` and thus the guess must match this criteria (because `2 < 3`). A guess of `[1, 5, 4]` should return `true` for this answer as `5 > 3`.
- `Verifier08` - "Number of 1s in the code".
    -
    - This verifier compares the number of `1`s in the answer to the number in the guess. E.g. an answer of `[2, 2, 2]` (no `1`s) compared to a guess of `[3, 4, 5]` (also no `1`s) returns `true`, whereas an answer of `[1, 2, 5]` compared to a guess of `[1, 4, 1]` should return `false` as the number of `1`s in the answer is 1, but in the guess is 2.
- `Verifier11` - "Checks the 1st digit compared to the 2nd (ie x<y, x=y or x>y)".
    - e.g. if the answer is `[4, 2, 5]` and the guess is `[1, 3, 2]`, the verifier should respond `false`, as `4 > 2`, but `1 < 3`.
- `Verifier21` - "If there is a number present exactly twice".
    - E.g. `[1, 3, 4]` and `[2, 2, 2]` don't have any pairs (`[2, 2, 2]` has *three* `2`s, not *exactly* two `2`s), however `[4, 3, 4]` does. So an answer of `[1, 3, 4]` and a guess of `[2, 2, 2]` should answer

`true`, but a guess of `[4, 3, 4]` instead would give the response `false`.

The `Machine`

The `Machine` class should have the following methods:

- a `public` constructor that takes an `int` array for the game answer and an array of `Verifier`s.
- a `public` constructor that takes an `int` array for game answer and a `List` of `Verifier`s.
- Both constructors must check that the `int` array is a valid answer (3 digits, all in the range from 1-5 inclusive), and invalid answers must throw an `IncorrectDataException`.
- A `toString` method that overrides the standard `toString` and returns a `String` representing the list of available `Verifiers`. Details on the format of this are given below.
- a `public turn` method that takes an `int` array as a guess and an array of `char` verifier choices, and returns a `String` describing the results of the guess. It throws an `IncorrectDataException` if the verifier does not exist, or the guess is out of bounds, or if the number of verifier checks isn't equal to 3. Details on the format of this are given below.

- a `public` `turn` method that takes an `int` array and a `String` of comma-separated verifiers and uses the same logic as `turn` described above.
- a `public` method `finalGuess` that takes an `int` array, and returns `true` if the input matches the answer. It should throw an `IncorrectDataException` if the guess is out of bounds.

You will need to add `private` fields to help you complete these methods, particularly a data structure that holds the answer, and one to store the `Verifiers`.

Notice again that the verifier `char`s should be the first n letters from `A`, `B`, `C`, etc., where n is the number of `verifier`s, Your code should handle both upper and lower case letters in the input.

You can also add other methods, but they should be `private`.

What does everything **look** like?

When a `Machine` is converted to a string it should have the basic format:

```
Verifiers:
A) Compares the 2nd digit to the value '3' (is it < 3, = 3 or > 3?)
B) Number of 1s in the code
C) Checks the 1st digit compared to the 2nd (ie x<y, x=y or x>y)
D) If there is a number present exactly twice
```

As the combination of `Verifier`s that a particular `Machine` has will change from test to test, you cannot hard-code this list, the above is just an example.

The `turn` function should look like this, given an input of `"a,D,c"` (for example):

```
Results for guess 122:
A) true
C) true
D) true
```

Notice that each letter is in uppercase, despite the input given, and they are sorted in alphabetical order (so `"A,C,D"` and `"d,A,C"` would both return the same as above)