Programming Assignment #1 CS671

1st submission due 6/9/2021
2nd submission due 6/16/2021

This first programming assignment is a "get your feet wet" exercise in Scala. It makes limited use of Scala's powerful features and can be implemented following a mostly imperative, Java style of programming. The first two sections specify practice problems that are used to prepare for the main assignment. Students are required to implement them as an exercise, but they can choose different patterns when implementing the main feature.

Simple Object-Oriented Programming

This section implements a simple type `CountingRef` to keep track of how often a value is used in a program. The idea is to wrap the value in a reference and to count how many times the reference is queried for the value. The type defined here is of limited use and is only presented for practice. However, the concept of reference classes can be useful in general (see `SoftReference` or Java's `AtomicReference`, for instance).

The trait that defines the reference type has a type parameter (the type of the value) and specifies the following public elements:

- `get`: This returns the value inside the reference. It also increments the access count as a side effect.
- `accessCount`: The number of times method `get` has been called since the reference was created (or since the last reset). The method has no side effect.
- `reset`: Resets the access count to zero.
- `disable`: Disables the reference. Methods `get`, `accessCount` and `reset` become disabled and throw `IllegalStateException`. Disabled references cannot be reenabled. This method has no effect if the reference is already disabled.
- `toString`: This returns the string representation of the value inside the reference. This method remains available after a reference has been disabled.

Trait `CountingRef` is provided. Students must implement class `CountingRefImpl`, which extends the trait. This class must implement all the methods of the trait with their names and signatures unchanged. It can use other methods and fields, but they should not be public. Note that at least one field is needed to keep track of the access count. The primary constructor of the class takes the value to be wrapped inside a reference. Null values are rejected with an `IllegalArgumentException`.

Loops and Lists

Lists are the most fundamental data structure in Scala, and can be used when implementing the last part of the assignment (below). This section defines several exercises that implement patterns that may be useful in the `Mastermind` class (but students are free to choose other patterns when implementing the class).

Some of these functions can be implemented elegantly using functional programming techniques (recursion, pattern-matching, higher-order functions, etc.). Functional programming is covered later in the semester. For this assignment, imperative implementations are acceptable, even if inelegant. Everything in this section can be done using `while`, for or `for`/`yield` loops. No recursion is needed.

- `find42(rand)`: Repeatedly calls method `nextInt` (with no argument) on `rand` until it returns 42. The function returns the number of times method `nextInt` was called (which is at least one).
- `findBig(rand)`: Repeatedly calls method `nextInt` (with no argument) on rand until it returns a number larger than 1,000,000. The function returns this number.
- `removeLongerThan(list, n)`: Removes from the list all the strings that have more than `n` characters. The function returns a new list containing the remaining strings, in their original order.
- `makePairs(list)`: Transforms a list of strings into a list of pairs. The first element of each pair is the corresponding string in the input list; the second element is the length of that string.
- `nextLetter(rand)`: Produces a pseudo-random uppercase letter. It does so by calling `nextInt(26)` on `rand` and by mapping the resulting number to a corresponding letter: 0 is A, 1 is B, . . . and 25 is Z.
- `makeLetters(rand, n)`: Relies on `nextLetter` to build a list of `n` pseudo-random letters. Function `nextLetter` is called exactly `n` times. The letters appear in the list in the order in which they were produced.
- `getIfValid(ref)`: Returns the value inside the reference as an option, unless the reference has been disabled, in which case it returns `None`. Note that there is no method in the `CountingRef` trait to test if a reference is disabled or not. Instead, one needs to call method `get` and handle the exception if the reference is disabled.
- `makeRefs(list)`: Wraps all the elements of the list into references and produces of list of references, in the same order.
- `getRefs(list)`: Extracts the values from the references in the list and returns them as a list, in the same order. (This is the inverse of `makeRefs`.) The behavior of this function is unspecified if the list contains a disabled reference.
- `getValidRefs(list)`: This function is similar to function `getRefs`, but skips all disabled references. The remaining values are returned in order.

Mastermind

For this part of the assignment, you will be writing code to help users learn to become better guessers in the game of Mastermind. The site below briefly talks about the game and its rules, and also provides a way to play the game on the page: https://www.archimedes-lab.org/mastermind.html. There is a `MastermindApp.scala` file that is not ready yet, but it gives you a sense for how you might expect the methods you will implement would be run. You should also look at some of the tests in `SampleMastermindTests.scala` to get some sense of how the methods might get called.

You must implement the following methods in `Mastermind.scala`:

- Constructor: The constructor must generate an `IllegalArgumentException` if the passed color list does not contain between 2 and 10 colors, or if there is a code passed that is not between 2 and 7 characters long. It should also generate a code of length 4 if no code is passed. You will probably also want to initialize a list with all possible codes that can be used for `validate`, `makeGuess`, `generateGuesses`, `analyzeGuess`, and `genereateBestGuess`. To do this can involve a call to `generateAllPossibilities`.

- `generateAllPossibilities`: Generates a list of all possible guesses, either as a list of strings, or as a list of lists of characters (whatever you prefer to use internally). A suggestion for how to build this list without using recursion:
    - start from a list with one empty string (or empty list)
    - repeat the following for as many characters are in the secret code:
        - generate all possible combinations of the available colors with each item from the list, and store it back into your original list.
- `numCodesRemaining`: Return how many possible codes remain, given the guesses already made. As calls to `makeGuess` are made, this number should decrease.
- `validate(guess)`: Determine if the passed `guess` is valid given the guesses made already. Returns `true` if consistent with all guesses made so far, or `false` if contradicted by evidence from prior guesses.
- `evaluateGuess(guess, code)`: Return the evaluation string for the passed `guess` and `code`. The return value contains an 'O' character for every color in the guess that is in the right place in the code, and an 'X' character for every color in the guess that is not in the right place, but is present in the code. All 'O' characters come before any 'X' characters.
- `evaluateGuess(guess)`: Return the evaluation string for the passed `guess` and the internally tracked secret code selected when the Mastermind object was created. It is fine for this method to call the two-argument method.
- `makeGuess(guess)`: Add the passed `guess` to the lists of guesses made already and return the result of `evaluateGuess` (one-argument version) on the passed guess. This should update an internal representation of what the possible secret codes could be, given the feedback returned for all the guesses made.
- `generateGuess`: Randomly generate a guess from current list of possible secret codes
- `analyzeGuess(guess)`: Return the percentage of valid guesses that the passed `guess` would eliminate in the worst case, given the list of guesses made already. This must be done by evaluating the `guess` against each code in the current list of possible secret codes. Think of it as a partition (Map) that indicates how many codes would yield “`XX`” for the `guess`, how many would yield “`OOX`”, how many would yield an empty string, and so on.
  So if you're starting off with no guesses made, you would expect to get this partitioning for a guess of “`YKYK`”:

`(,256)`	`(OX,208)`	`(OXX,36)`	`(OOXX,4)`
`(O,256)`	`(OO,114)`	`(OOX,32)`	`(OOOO,1)`
`(X,256)`	`(XX,96)`	`(OOO,20)`	`(XXXX,1)`
		`(XXX,16)`	

  The categories with the most potential codes in them are “”, “`o`”, and “`X`”, with 256 each. With 6 colors and 4 positions, there are 1296 total possibilities. Subtract 256 and you are left with 1040 guesses eliminated in the worst case.
- `generateBestGuess`: Generate best guess from current possibilities based on the percentage of guesses it would eliminate. This should be built off the information you get from `analyzeGuess`, possibly on many repeated calls.
- `solved`: Return true if one of the guesses matches the code, false otherwise

Notes

- Source code is submitted for testing using Git (master branch). The first submission must be tagged 1a and the second submission 1b (even if the first submission is missing). If modifications to the code are needed after tagging a submission, a new tag should be created of the form 1a.1, 1a.2, 1b.1, etc. Do not delete tags!
- In addition to this Git submission, students need to upload on MyCourses a sample of their code for review. Code should be uploaded as a pdf file created from a pretty-printer (e.g., a2ps, pandoc, highlight, IntelliJ, etc.). The sample should be about one page long, and can include specific questions (e.g., This code is very convoluted; could it be made simpler? or This code seems to be very slow and I don't know why or I'm very proud of this code, which follows a pure functional pattern; what do you think?).
- API details of the classes and objects to implement ***will eventually be*** in the 1/api directory of the Git repository.
- Students should start with the implementation of the CountingRefImpl class and the exercises in the practice package. These implement patterns that can be used when implementing Mastermind. There is, however, no requirement to use any of those patterns in the Mastermind class, which can be implemented as students choose.
- There is no programming style requirement for this assignment. Students can choose a functional or an imperative style as they prefer. However, several functions of the practice part are more easily implemented using for/yield than with while loops.
- The command-line game application code is given in MastermindApp. While not fully implemented, it can be freely modified for testing and debugging purposes. It will not be used in grading tests.
- Sample tests can be run from IntelliJ IDEA or from the command-line using sbt:
  ```
  > testOnly SampleTests
  ```
  or
  ```
  > testOnly SampleTests -- -oD
  ```
  to show test durations. It is not recommended to use the sample tests as debugging tests. Students should write their own tests for this purpose.
- This specification is not yet complete. More notes may be added soon.