COM1001 Introduction to Software Engineering

Ruby Programming With Style

Ruby concepts introduced in this lab: *Coding conventions* · *Ruby Style Guide*

1 Ruby Style Guide

We have already seen that Ruby has a very flexible syntax, and there are many different ways in which the same code can be written. However, it is important to agree on certain conventions and best practices, such that the code you write can easily be read, maintained, and re-used by other Ruby programmers. Such conventions and best practices are usually summarised in *style guides*, and there is also a community effort to create such a style guide for Ruby. You can find the Ruby style guide at the following URL:

`https://github.com/bbatsov/ruby-style-guide`

You can also find further examples at `https://github.com/styleguide/ruby`.

The style guide covers a range of different aspects of what is considered good Ruby code. Here are some examples:

1.1 Formatting

- Indent code with two spaces.
- Keep lines shorter than 80 characters.
- Never leave trailing whitespace.
- End each file with a blank newline.
- Use spaces around operators, after commas, colons and semicolons, around { and before }.
- No spaces after (, [or before],).
- Use empty lines between method definitions (`def`).

1.2 Syntax

- Use `def` with parentheses when there are arguments. Omit the parentheses when the method doesn't accept any arguments.
- Don't use `for` loops.
- Don't use `then` for multi-line `if`/`unless`.
- Don't use the `and` and `or` keywords, but always use `&&` and `||` instead.
- Never use `unless` with `else`.
- Don't use parentheses around the condition of an `if`/`unless`/`while`.
- Never put a space between a method name and the opening parenthesis.

1.3 Naming

- Use `snake_case` for methods and variables.
- Use `CamelCase` for classes and modules.
- Use `SCREAMING_SNAKE_CASE` for other constants.

2 Hangman

Today's exercise is a simple word guessing game. The word to guess is initially represented by a row of dashes, one for each letter of the word. If the guessing player suggests a letter which occurs in the word, it is written in all its correct positions instead of the dashes. If the suggested letter or number does not occur in the word, the game adds one element of a hanged man stick figure as a tally mark. If the hanged man is complete, the player loses.

3 Basic Guessing Game

We will start by implementing a basic version of the main game loop, where we simply count the number of attempts. We will add the stick man and other features later. The following description walks you through one possible way to implement the hangman game. If you think you can do it without these instructions, feel free to take your own path to the solution.

Let's begin by defining some variables we will need for the main loop:

1. We need an array of words. For now, just create an array such as, for example:

   ```
   words = ["COM1001", "Hello", "Sheffield"]
   ```

2. Select a random entry from this array and assign it to a variable, e.g. `word`. (Recall that you can get a random index using `rand(word.length)`, or you can simply sample a random element from the array by using `words.sample`).

3. We somewhere need to store information about which characters have already been successfully guessed by the player. One way to do this is by using an array of the length `word.length`, initialised with "_":

   ```
   characters = Array.new(word.length, "_")
   ```

   The idea is that this array of characters is shown to the player so that they know which letters they have already guessed, and how many letters there are in the word in total. Thus, over time, the "_" characters will be replaced with the actual characters of the word.

4. We also need to keep track of the number of remaining attempts, e.g., `attempts = 9`.

3.1 Output Method

Create a method "`output`", which takes two parameters: 1) The number of attempts (`attempts`), and 2) the array of guessed characters (`characters`).

1. First, print the current guess. To do this, we need to iterate over the array of characters and print each on the screen. Recall that Ruby loops are preferrably written using the `each` structure:

   ```
   characters.each do |c|
     print c
   end
   ```

2. Inform the player about the remaining number of attempts, and prompt them to enter a new character (but let's do the actual reading of the input later).

3.2 Checking the Guess

Next, create a method "`is_solved`" with two parameters: 1) The actual word (`word`), and 2) the array of guessed characters (`characters`). The method should return the boolean value `true` if the characters match the word, and `false` if not. To do this, you will need to have a loop that iterates over the characters. In contrast to the previous method, however, here we need to iterate over the index, so

that we can compare characters one by one. Thus, create a loop over the range `(0..word.length-1)` using the `each` method, which will give you an index (e.g., `w`), and then you can compare the element in `characters` at position `w` (`characters[w]`), and the character in the string `word` at position `w`. To get the character at a specific position in a string, you can use the same syntax as when accessing an array, i.e., `word[w]`.

3.3 Updating the Guess

When the player has entered a new character, we need to update the array of guessed characters. Add a method "`update_and_check`" which takes three parameters: 1) The array of characters (`characters`), 2) the actual word (`word`), and 3) the new guess (`guess`). The method should return `true` if the array of correctly guessed characters (`characters`) was changed, i.e., when the guess was successful, and it should return `false` is the newly guessed letter does not occur in the word.

To determine whether the guess is successful, we need to iterate over the letters of the word, and for each letter check whether it matches the guess. If it does, then we need to update `characters` with this letter (to replace the "_" at that position). At the end of the iteration, we will need to know whether we have replaced any letters, so create a helper variable `found` before the loop, initialise it to `false`, and set it to `true` if a character is replaced. After the loop, simply return the value of `found`.

3.4 Main Loop

Now it is time to add the main game loop. After initialising `word`, `attempts`, and `characters` (see above), add a new loop that iterates while `is_solved(word, characters)` is false and `attempts > 0`. In this loop, do the following:

1. Call the output method `output`.
2. Read the user input, and store the result in a new variable.
3. Update and check `characters` based on the user input using the method `update_and_check`.
4. If `update_and_check` returns false, then we need to decrease `attempts`. If the guess was correct (i.e., the method returns `true`), then the number of attempts remains the same.

Finally, after we have exited the loop, we need to tell the player whether they have won or lost the game. Use the `is_solved` method to determine this.

4 Suggestions for Improvements

- Rather than just telling the player the number of remaining attempts, show the state of the gallows. On the MOLE page you will find a file `hangman.rb` which contains an array of 9 strings that show stages of the hangman. Include this array, and show the status at each iteration of the game, by selecting the correct entry of the array based on the value of `attempts`.
- Make the game case insensitive; that is, the player should only have to guess the letters, not whether they are upper case or lower case. In the word where the letters are revealed, however, the original case should be displayed.
- Make sure the player can only enter single letters. This could be done by checking the length of the input after removing the line break. (As an alternative, you could read only one keypress at a time. In Ruby, this is done using the command `STDIN.getch`, for which you need to include the io/console module: `require 'io/console'`. However, note that this will not work on the RubyMine Windows console.)
- On the MOLE page, you will find a dictionary of words. Download the file, put it in the same directory as your Ruby script, and read the contents to an array, just like we did last week with the file of quotes. Then, randomly chose one of these words to play.

- Refactor your program such that the main game loop is contained in its own method (e.g., "`game_loop`"). Call that method in a loop, and after each round ask if the player wants to play again. If the player responds that they do not want to play again, quit the game, otherwise start another round of the game.
- If you want to clear the screen at the beginning of `output`, such that the output always appears at the same position, then simply use the following snippet of code:

  ```
  puts "\e[H\e[2J"
  ```

  This is an ANSI escape code to clear the console, and it works in any console that supports ANSI characters (any Mac or Linux command line terminal will do that). However, as before with the `STDIN.getch`, this will not work in RubyMine on Windows.

5 Example

Here is an example how the game looks like with the hangman included:

```
   +---+
   |   |
   O   |
  /|\  |
       |
       |
 =========
The word is: COM1__1
You have 2 failed attempts left. What is your next guess?
```

If we guess correctly, the game ends and the player is informed of their victory:

```
   +---+
   |   |
   O   |
  /|\  |
       |
       |
 =========
It is COM1001, you win!
```

If the player guesses wrong 9 times, they are dead.

```
   +---+
   |   |
   O   |
  /|\  |
  / \  |
       |
 =========
You lose!
```