COM1001 Introduction to Software Engineering

More Pair Programming

Ruby concepts introduced in this lab: *Loops* · *Arrays* · *Random numbers*

1 Introduction

In this exercise, you will implement a simple interactive program that simulates a conversation with an old lady that is hard of hearing. The conversation is done via textual input/output, like this:

```
> Hello.
HUH? Speak up sonny!

> ARE YOU ALRIGHT?
No, not since 1952!

> BYE!
Oh ok, bye.
```

2 Pair Programming

For this lab, we will be using pair programming again. Like last time, find a partner to work with for this lab (or let us know if you can't find anyone). Refer to last week's lab sheet for details on pair programming. As a quick reminder:

- Partners sit side-by-side at one computer.
- The "driver" has control of the keyboard/mouse and actively implements the program.
- The "navigator" observes the work of the driver to identify tactical defects (such as syntactic and spelling errors, etc.) and also thinks strategically about the direction of the work.
- You should change roles every 10 to 15 minutes or at natural breaking points.

3 Loops in Ruby

There are several different types of loops in Ruby, some of them similar to what you already know from Java. The while loop takes a conditional expression (similar to an `if` statement), and executes everything in the code block from the `while` to the `end` until the condition becomes false. The `do` statement is not mandatory, but there needs to be something that separates the condition from the loop body (a `do`, a newline, a backslash, or a semicolon):

```
i = 0
while i < 5 do
  puts "Iteration #{i}"
  i +=1
end
```

It is also possible to have the condition at the end of the block, such that even if the condition is false from the beginning, the code block still gets executed once (like a do-while loop in Java):

```
i = 0
begin
  puts "Iteration #{i}"
  i +=1
end while i < 5
```

Another variation of the `while` loop is the `until` loop, which works similarly, except that the logic of the condition is inverted. That is, it executes the loop as long as the condition is false, and stops the loop once the condition is true:

```
i = 0
until i >= 5  do
  puts "Iteration #{i}"
  i +=1
end
```

There is also a `for` loop, which takes a variable and an iterable expression, such as a range:

```
for i in 0..4
  puts "Iteration #{i}"
end
```

The range expression `0..4` produces the numbers 0, 1, 2, 3, and 4. However, even though this loop is supported in Ruby, the preferred way to do this is using `each`:

```
(0..4).each do |i|
  puts "Iteration #{i}"
end
```

The `each` method is the common way to iterate over arrays, explained below.

Just like in Java, you can also influence the excution of loops using `break`. In the following example, the `for`-loop that is declared to go from 0 to 5 will in fact only run through 0, 1, and 2, and in the iteration where `i` equals 3 the `if` condition is true and the `break` will exit the loop before the next `puts` is executed:

```
for i in 0..5
  if i > 2 then
    break
  end
  puts "Iteration #{i}"
end
```

The equivalent of the Java `continue` command in Ruby is called `next`. If called, it goes back to the beginning of the loop body and starts the next iteration. For example, the following `for` loop will go through all values for `i` from 0 to 5, but the `puts` will only be executed when `i` is 2 or larger:

```
for i in 0..5
  if i < 2 then
    next
  end
  puts "Iteration #{i}"
end
```

4 Arrays

The second new concept in this exercise is arrays. An array is a list of elements, where each element can be addressed by its index in the list. The following creates a variable `arr` which is assigned an array consisting of three elements:

```
arr = [ 1, 5, "hello" ]
```

You can use this variable like any other variable, for example to pass it as a parameter to a method. To access the individual elements of the array, use square brackets and the index. The index of the first element in the array is 0. For exampe, `arr[0]` will return 1, `arr[1]` will return 5, and `arr[2]` will return "`hello`".

Ruby provides syntactic sugar with its arrays that is not available in Java. For example, to access the last element, you can use the index -1; to access the penultimate element use -2, and so on. You can also assign several elements at the same time using ranges (e.g. `arr[0..4]` returns an array that contains the first 5 elements of `arr`), concatenate arrays using the + operator, and many others.

If you want to create an array programmatically (i.e. you do not know the values while writing the code, but want to assign them later), you can create an array using the `Array.new` command, and then assign values to indices. For example, the following creates an array of length 10, and assigns the value 0 to all elements; then we set element 5 to the string "`hello`":

```
arr = Array.new(10, 0)
arr[5] = "Hello"
```

As mentioned earlier, the `each` statement is the preferred way to iterate over arrays (although in principle you could also use a traditional `for` loop from 0 to the length, which you can query with the method `length`):

```
arr.each do |element|
  puts "Next element of the array: #{element}"
end
```

5 A Basic Old Lady

Let's start the programming exercise with a basic version of the old lady. Start RubyMine (or your favourite Ruby editor) and create a new project. Add a file `oldlady.rb`, and implement the following:

1. In a loop:
   a) Output a prompt (e.g. >).
   b) Read input from the user.
   c) Output a response.
2. Initially, let the response be "`What?`".
3. The old lady only stops talking when you shout "`BYE!`"; that is, the loop only terminates upon this input.

6 Some More Interaction — Random Numbers

Now let's make the responses a bit more interesting by adding some variation. We will achieve this by using the random number generator of Ruby: To generate a random number, use the method `rand`. If you invoke `rand` without parameter, it will return a random floating point number in the range $[0, 1]$. If you add a parameter, then this defines the range of integer numbers. For example: `rand 10` returns a random integer in the range $[0, 10)$.

1. Change the somewhat boring response "`What?`" to randomly either "`What?`" or "`HUH? Speak up sonny!`". For example, pick a random number in some range, and if the random number is less than half of that range then output one, else output the other statement.
2. Shouting at the old lady should result in a different response. Check if the message entered by the user is in all caps (which means we're shouting, but not "`BYE!`"). If so, then let her respond with a random quote of the type "`No, not since <random number between 1930 and 1950>`". There are different ways to check if a string is in all caps; a simple way is to create a version of the string that is in all caps (method `upcase`) and compare if that is the same as the original string.

7 A Stubborn Old Lady — Counting in Loops

The old lady doesn't want you to leave. If you say `BYE!` she will pretend not to hear you (use the usual response), and you can only leave by shouting `BYE!` three times in a row. If you shout `BYE!` three times, but not in a row, you need to continue talking to the old lady.

8 A Chatty Lady

Every old lady knows old stories that she likes to tell. Download the file `quotes.db` from the MOLE page (in the labs section). In this file, there is one quote per line, and we can simply read the file to an array of strings (e.g., in a variable called `quotes`) using:

```
quotes = File.readlines "quotes.db"
```

That is, `quotes` now is an array of strings, and each entry of the array is a quote. Note that the file needs to be in the same directory as the the Ruby script `oldlady.rb`.

When entering a statement that is not a question and not `BYE!`, let the old lady answer with a random quote selected from the array of quotes read from the file. There are different ways to achieve this; you could use a random index (e.g. `quotes[rand(quotes.length)]`), or you could use the `sample` method of arrays (`quotes.sample`) which returns one randomly chosen element from the array. (Check RubyDoc for details: `http://ruby-doc.org/core-2.2.0/Random.html`)

9 Too Easy?

If this was too simple for you, here's some suggestions to make the old lady more interesting:

- Add a timer to the conversation: If there is no input for 5s, let the old lady say a quote by randomly choosing one of the quotes from the array.
- Once the conversation is finished, it would be interesting to print out some statistics:
  - How long did the conversation last? (See `Time` class).
  - How many times did you have to say `BYE!` in total?
  - How many quotes did the old lady say?
  - What was her favourite quote?