Çagatay Safak - 21902730

CS315 - HOMEWORK 3
SUBPROGRAMS IN RUBY

Introduction

Subprograms, in a broad sense, are small programs which will be used by the larger programs. Their purpose is to simplify the whole process by defining it by steps. This increases readability and writability of a program, since when we use subprograms, we have to write only one line of code instead of writing the content of the subprogram each time.

Another benefit of using subprograms is the easier debugging. By using subprograms, we can detect where the code flow is broken easier. It is easier to test subprograms also. This reduces the time needed to finish the code for developers.

Moreover, we can use a subprogram in other code files, by including them. For example, all the libraries, APIs, etc. use the same logic. Abstraction is one of the most important

concepts in programming, and by using subprograms, we can implement abstraction perfectly.

In this homework, how the Ruby language deals with subprograms will be investigated.

Evaluation of the language

In the Ruby language, subprograms (methods) are quite similar to the ones in other programming languages. In Ruby, if something starts with a capital letter, the interpreter thinks that it is a constant. Since we do not want to build our methods as constants, they start with lowercase letters. Also, all methods should be defined before calling them, like a majority of programming languages. It would be useful to note that the implementing language of Ruby is the C language.

Furthermore, Ruby is a dynamically typed language, which means that the interpreter tries to find the declarations at run-time. On the one hand, this increases the speed of a developer coding, since it makes the programs more dynamic and easier. On the other hand, this makes debugging so much harder, since it is more difficult to detect where the error occurs in dynamically typed languages.

Design issues

Almost every modern programming language includes a subprogram structure, but each of them handles it in a different way. Here is how it works in the Ruby language.

Subprogram overloading

Subprogram overloading (method overloading) is a case of a class with more than one subprogram with the same name, but different signature.

The Ruby language does not support subprogram overloading, since it is a dynamically typed language. As we discussed in the “Evaluation of the language” part, in dynamically typed languages, the interpreter tries to find the declarations at run-time. Also, the origin of Ruby is the C language, and C does not include subprogram overloading in the first place. If one tries to overload subprograms in Ruby, the interpreter will accept the last occurrence of the subprogram, instead of checking their signatures. Here follows an example.

```ruby
class MyClass
    def self.sum(a,b)
        puts(a+b)
    end

    def self.sum(a,b,c)
        puts(a+b+c)
    end

    def self.identity(*args)
        case args.size
            when 1
                puts "NAME =  #{args[0]}"
            when 2
                puts "NAME =  #{args[0]} SURNAME = #{args[1]}"
        end
    end

end

puts "---SUBPROGRAM OVERLOADING---"

#MyClass.sum(1,2) <--- GIVES ERROR
MyClass.identity "Cagatay"
MyClass.identity "Cagatay", "Safak"
```

As it can be seen, the program gives an error when we try to reach an overloaded method. Also, it can be seen that there are a few ways to implement subprogram overloading manually in the Ruby language. Here is the output

```
$ruby main.rb
---SUBPROGRAM OVERLOADING---
NAME =  Cagatay
NAME =  Cagatay SURNAME = Safak
```

Return values

In the Ruby language, subprograms always return only one thing. In other words, there is no void method in Ruby. Even in void cases, subprograms return an object called **`nil`**, which means nothing but is still an object. Ruby includes the **`return`** keyword, but unlike other languages, it is not mandatory to use **`return`**. In Ruby, if there is no returning statement in a subprogram then the subprogram returns the last evaluated statement in the body of the statement, which is usually the last line. Here is a snippet of a code that includes this design issue.

```ruby
class MyClass
    def self.increaseBy(no)
        no + 1
        no + 2
        no + 3
        no + 4
    end
end

puts "---RETURN VALUES---"

puts "THE INCREMENTED NUMBER = #{MyClass.increaseBy(30)}"
```

Here is an image of the output of the program above. As it can be seen, the value of the last line of code inside the subprogram has been returned.

```
$ruby main.rb
---RETURN VALUES---
THE INCREMENTED NUMBER = 34
```

Nested subprogram definitions

It is possible to define nested subprograms in Ruby, however, there is no complex structure like the ones in Python. Ruby language has most of these complexities in the structures like lambdas, blocks, and procs.

In the Ruby language, when two functions are nested, we cannot access the nested one directly. First, we need to call the outer one in order to make its execution environment accessible. Then we can access the inner definition.

Keywords and default parameters

After Ruby 2.0, it is possible to use keyword parameters for subprograms. Here is my code and its output.

```
class MyClass
    def self.hiddenTaxCalculator(income, taxRate)
        income*taxRate
    end

    def self.publicTaxCalculator(income:, taxRate:)
        income*taxRate
    end
end

puts "\n---KEYWORDS AND DEFAULT PARAMETERS---"

puts "HIDDEN TAX = #{MyClass.hiddenTaxCalculator(5000, 0.20)}"
puts "PUBLIC TAX = #{MyClass.publicTaxCalculator(income: 5000,
taxRate: 0.20)}"
```

Output:

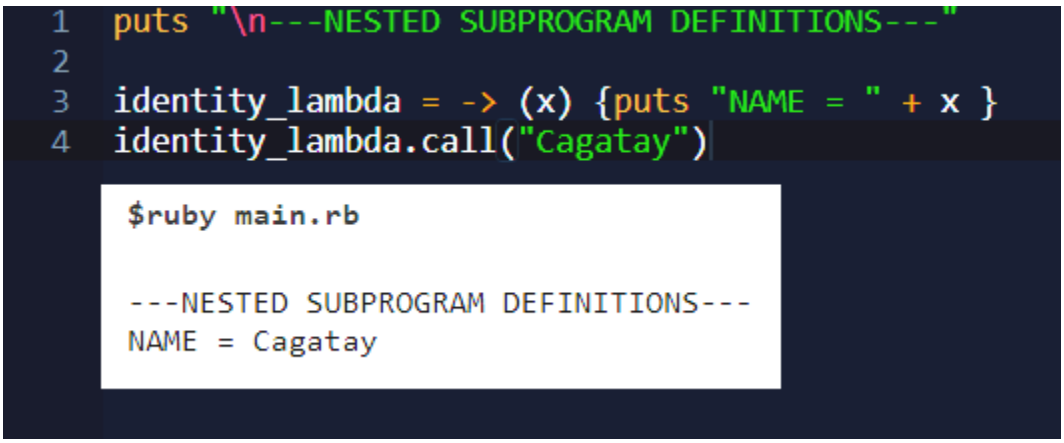
```
---SUBPROGRAM OVERLOADING---
NAME =  Cagatay
NAME =  Cagatay SURNAME = Safak

---RETURN VALUES---
THE INCREMENTED NUMBER = 34

---KEYWORDS AND DEFAULT PARAMETERS---
HIDDEN TAX = 1000.0
PUBLIC TAX = 1000.0

---CLOSURES---
NAME = Cagatay
```

Closures

Closure is a subprogram and all the referencing environment at which this subprogram is defined. Since closures are only required for the subprograms which

are able to access variables in nesting scopes, static-scoped languages, which do not permit nested subprogram definitions, do not need closures.

In the Ruby language, blocks, lambdas, and procs are closures. Here is an example code for the lambda structure in Ruby, with its output.



My learning strategy for the topic

I have learned the topic from:

- http://ruby-for-beginners.rubymonstas.org/writing_methods/return_values.html
- https://www.geeksforgeeks.org/method-overloading-in-ruby/
- https://betterprogramming.pub/static-typing-in-ruby-3-0-ba46f43a7f3a
- https://thoughtbot.com/blog/ruby-2-keyword-arguments
- The textbook.

In order to write the code and see the result, I used an online compiler which is called TutorialsPoint.