Subprograms in Ruby

In this homework, I tested all design issues in Ruby 3.

1. Subprogram overloading

According to our textbook, Ruby supports both method overloading [1]. However, several online sources say that there is no subprogram overloading in Ruby. When I examine it myself, I have realized that since Ruby has dynamic typing (and there are no keywords to assign a type to a variable), it is not possible to overload a method by changing the variable type, for example, there are no such definitions in Ruby: foo(int a) or foo(string a), hence it is not possible to overload a subprogram by changing the types of arguments. Moreover, it is possible to have two methods foo(a) and foo(a, b); however, when there are two functions with the same name but different sets of parameters, the latter defined function prevails [9].

```
def berk(a)
      puts a
end
def berk(a, b)
      puts a+b
end
berk(5,3)
# berk(5) # Error!

Output:
8
```

In the above example, Ruby ignores the first subprogram definition with the single argument, and it only considers the second definition; therefore, it gives an error when we try to call the function with only one parameter. It is also possible to define a function with the optional number of arguments and return different values according to the parameter count. However, this is not the method overloading that we know from Java.

To conclude, it can be said that there is method overloading to some extent since it is legal to have functions with the same name but different parameters; however, the latter function invalidates the previous functions. Moreover, this method overloading (if we can call this method overloading) is different from Java since we cannot use both functions at the same time.

2. Return values

In Ruby, functions always return values, if it is not specified which line to return, functions automatically return the last value of the program.

```
def foo()
    str = "I want to return this!"
    str
    "HAHA"
end
puts foo()

Output:
HAHA
```

In the above example, function foo returns the last value which is “HAHA” since there is no return statement [3].

```
def foo2()
    str = "I want to return this!"

    return str
    "HAHA"
end
puts foo2()

Output:
I want to return this!
```

Unlike the prior example, this time str will be returned since this time return statement is used.

```
def foo3(b, s, y)
    return b*s*y, "berk", 1999, [3,4,5], {"berk"=>"I cry myself to sleep",
"burcu"=>"Hi!"}, :bsy  # As well as returning only 1 value as in foo2, it is
possible to return multiple values
end

puts foo3(2,3,4) # It prints aaaaall values returned from this function
# It is also possible to access elements one by one!
puts "Element in index 1: #{foo3(2, 3, 4)[1]}"
puts foo3(2,3,4)[3][2]
puts "#{foo3(2,3,4)[4]['berk']}"

Output:
24
berk
1999
3
4
```

```
5
{"berk"=>"I cry myself to sleep", "burcu"=>"Hi!"}
bsy
Element in index 1: berk
5
I cry myself to sleep
```

In Ruby, it is possible to return multiple values as an array. In the above example, the function returns the multiplication of function parameters, a string, an integer, an array, a hash, and a symbol. Furthermore, it is possible to access individual return values with array subscript. For example, it is possible to access the string value returned, the third element of the returned array, and even it is possible to access the hash value with the key “berk”. This provides flexibility and it is useful when a function needs to return multiple things. However, if we did not know the inner structure of the function, it would be confusing.

```
def dummy()
    return
end
puts dummy() # This prints literally nothing :(
yummy = dummy()
puts yummy.inspect # Type of yummy is nil…

Output:

nil
```

In Ruby, the empty return statement is allowed but if we assign the returned value to a function it takes the value nil.

```
def fact(a)
    if(a == 1)
        return 1
    end

    return a*fact(a-1)
end

puts fact(6) # Classic example of recursion!

Output:
720
```

Of course, it is possible to have recursion in Ruby by making a function call itself and returning it.

3. Nested subprogram definitions

In Ruby, it is possible to define nested subprograms.

```
def bilkent
    def kills
        def me
            puts "In me!"
        end
        puts "In kills!"
        me()
    end
    def hello
        def there1
            puts "In there1"
            me()
        end
        def there2
            puts "In there2"
        end
        puts "In hello"
        there1()
        there2()
        kills()
    end
    puts "In bilkent!"
    # me() # When me is called before kills, it gives an error since it is not
in call stack
    kills()
    hello()
    me()
    there2()
end
puts "In main!"
bilkent()

Output:
In main!
In bilkent!
In kills!
In me!
In hello
In there1
In me!
In there2
In kills!
In me!
```

```
In me!
In there2
```

In the above example, subprograms call each other without error. Moreover, it is possible to call the nested subprograms from outside but for this, the called subprogram must have been called once. For example, It is possible to call the “me” subprogram which lies inside the “kills” method from the main program; however, to do this first we have to call the “kills” method since it calls the “me” method and makes it visible from the other subprograms. Therefore, we can say nested subprograms are allowed but we cannot observe a static parent-children structure between the outer and inner functions unlike languages such as JavaScript, and Perl.

4. Scope of local variables

In Ruby, a variable becomes local if the variable starts with [a-zA-Z] or with _.

```
jax = "WOOF!"
puts jax

tiny = "mini woof!"
def bark()
    jax = "MEOW!" # This variable is local to the subprogram!

    puts jax
    # puts tiny # Error!

end
bark()
puts jax

Output:
WOOF!
MEOW!
WOOF!
```

In the above example, “jax” and “tiny” are the local variables of the main program first. When the bark subprogram is called, this program creates its own local variable “jax” and assigns it another value. Therefore, jax variables are different variables in main and bark. The value of the jax stays the same after the subprogram call. Moreover, if we would try to call an outside local variable, in this case, “tiny”, it would give an error because “tiny” that is a local variable of main is not visible from the bark subprogram.

```
def luigi()
    mainCharacter = "I am Luigi!"
    def itsAMeMario()
        # puts mainCharacter  # Error!
```

```
        mainCharacter = "It's-a me Mario"
        def princessPeach()
            mainCharacter = "Your princess is in another castle!"
            puts mainCharacter
        end
        puts mainCharacter
        princessPeach()
        puts mainCharacter
    end
    puts mainCharacter
    itsAMeMario()
    puts mainCharacter
end
luigi()

Output:
I am Luigi!
It's-a me Mario
Your princess is in another castle!
It's-a me Mario
I am Luigi!
```

As in the prior example, each subprogram creates its own local variable when the new value is assigned to the mainCharacter variable. Nested subprograms inside a subprogram also cannot see the local variables of static parents.

```
def funko(pop)
    pop = 13
    puts pop
end
pop = 17
puts pop
funko(pop)
puts pop

Output:
17
13
17
```

The same thing applies to function arguments too. When we assign another value to a function argument, it does not modify the variable in the callee environment.

```
jax = "WOOF!"
puts jax
1.times do |jax|
```

```
    jax = "MEOW!" # This is local to this block
    puts jax
end
puts jax

Output:
WOOF!
MEOW!
WOOF!
```

In Ruby, it is also possible to create a local scope by using blocks. Blocks are normally used for closure structure, and a block might be defined with do…end or {}. In the above example, the lines between do and end are executed 1 time, and it takes jax as the parameter, when another value is assigned to the variable, it becomes a reference to another object; therefore, it becomes a local variable to the block and the variable in the callee environment is not affected.

5. Parameter passing methods

According to our textbook, Ruby has pass-by-assignment. We can say each variable is a reference to objects that we put in them. When we say a = 33, this means a is a reference to the memory block that holds 33, we can represent this as a -> [33].

For example when both values are not changeable (in other words immutable). When they are passed in functions, the objects that are referenced by variables will not change at all. It can be said that if the referenced object is mutable, Ruby changes the value of the referenced object. However, in the example below both referenced objects are immutable. Therefore, x references another object after the assignment operator, and the value in the main does not change.

```
def adder(x, y)
    puts "in adder: id of x #{x.object_id}, id of y #{y.object_id}"
    x = x + y
    puts "in adder: id of x #{x.object_id}, id of y #{y.object_id}" # address
of x changes
end
a = 33
b = 44
puts "before subprogram: id of a #{a.object_id}, id of b #{b.object_id}"
adder(a, b)
puts "#{a}, #{b}" # 33, 44
puts "after subprogram: id of a #{a.object_id}, id of b #{b.object_id}"

Output:
before subprogram: id of a 67, id of b 89
```

```
in adder: id of x 67, id of y 89
in adder: id of x 155, id of y 89
33, 44
after subprogram: id of a 67, id of b 89
```

It can be seen that in the adder function, variables x and y are referencing the same objects as a and b. However, after the assignment operator, x is pointing to another object. Therefore the value of a in the main programs does not change!

```
def stringAdder(a, b)
   puts "in stringAdder: id of a #{a.object_id}, id of b #{b.object_id}"
   b << a
   puts "in stringAdder: id of a #{a.object_id}, id of b #{b.object_id}"
end
c = "bond"
d = "james"
puts "before subprogram: id of c #{c.object_id}, id of d #{d.object_id}"
stringAdder(c, d) #bond, jamesbond
puts "#{c}, #{d}"
puts "after subprogram: id of c #{c.object_id}, id of d #{d.object_id}"

Output:
before subprogram: id of c 60, id of d 80
in stringAdder: id of a 60, id of b 80
in stringAdder: id of a 60, id of b 80
bond, jamesbond
after subprogram: id of c 60, id of d 80
```

In the above example, objects referenced are mutable hence the values in the main program are changed and a and b are still referencing the same objects after the concatenation operation.

6. Keyword and default parameters

In Ruby 3, there are both default and keyword parameters.

```
def blackCoffee( coffeeBeans:, milk:"no milk", flavour:"no flavour")
    puts "This coffee has beans: #{coffeeBeans}, milk: #{milk}, flavour:
#{flavour}"
end
# blackCoffee() # Error!
blackCoffee(coffeeBeans:"arabica") # Default values!
blackCoffee(coffeeBeans:"guatemala", milk:"soy milk", flavour:"hazelnut") #
We can overwrite the default values
blackCoffee(milk:"cow milk", coffeeBeans:"ethiopia") # keyword arguments!
```

```
Output:
This coffee has beans: arabica, milk: no milk, flavour: no flavour
This coffee has beans: guatemala, milk: soy milk, flavour: hazelnut
This coffee has beans: ethiopia, milk: cow milk, flavour: no flavour
```

To make a parameter keyword parameter, we should add a colon at the end of this variable. After making the parameters keyword parameters, we can assign them default values. In the above example, all the parameters are keyword parameters but only two of them have default values. If we only pass one argument, the coffee beans argument takes the value and others take default values. However, we cannot call this function without giving any arguments because coffeeBeans does not have a default value.

```
def blackCoffee2( coffeeBeans, milk:"no milk", flavour:"no flavour")
    puts "This coffee has beans: #{coffeeBeans}, milk: #{milk}, flavour:
#{flavour}"
end

blackCoffee2("guatemala", flavour:"hazelnut", milk:"soy milk")
blackCoffee2("instant coffee")
# blackCoffee2() # Error!
Output:
This coffee has beans: guatemala, milk: soy milk, flavour: hazelnut
This coffee has beans: instant coffee, milk: no milk, flavour: no flavour
```

In the above example, two arguments are keyword arguments whereas coffeeBeans is a positional argument. To give values to keyword arguments, first, we should assign values for positional arguments. After that, we can assign values for keyword arguments in the order we want. It would give an error if we tried to give values to keyword arguments before the positional arguments. Also, it would give an error if we called the function with 0 arguments since coffeeBeans is positional and does not have a default value.

```
def beverage(drinkType, *bases, **flavours)
    puts "Drink type is: #{drinkType}"
    puts "Bases are: #{bases}"
    puts "Flavours are: #{flavours}"
end
     #drinkType,|    bases[],  |   flavours{key->value}
beverage("tea", "milk", "water", "flavour1"=>"hazelnut") # first one is
drinkType, other values are converted in an array until the key-value pair

Output:
Drink type is: tea
Bases are: ["milk", "water"]
Flavours are: {"flavour1"=>"hazelnut"}
```

In the above example, we see that it is possible to give arguments without a specified number for them. The first formal parameter of the beverage function takes a single value, the bases argument takes several actual arguments as an array and the last argument which is flavours takes a hash as key-value pairs.

7. Closures

There are three types of closures in Ruby: blocks, procs, and lambda.

```
favShows = ["Dexter", "BrBa", "The Office"]
# It is possible to create block closures with both {} and do...end
favShows.each { |show|
puts show
}

Output:
Dexter
BrBa
The Office
```

The above example shows the closure with blocks. “each” function passes the function given as the block for each element of the array. This means each element is being the parameter “show” of the function and their value is printed. It can be said that the block is treated as an anonymous function.

```
def geralt
    puts "Hey Roach, come here!"
    yield
end
geralt { puts "NEIGH!!!!!" }

Output:
Hey Roach, come here!
NEIGH!!!!!
```

Another usage of blocks for closure is demonstrated in the above example. A function inside the block is passed to the function called geralt and this passed function is called when the yield statement is encountered.

```
whatDoesTheFoxSay = Proc.new { |str| "#{str}"}

puts whatDoesTheFoxSay.call("Ring-ding-ding-ding-dingeringeding!")
puts whatDoesTheFoxSay.call("Wa-pa-pa-pa-pa-pa-pow!")
puts whatDoesTheFoxSay.call("Hatee-hatee-hatee-ho!")
puts whatDoesTheFoxSay.call("Joff-tchoff-tchoffo-tchoffo-tchoff!")
```

```
Output:
Ring-ding-ding-ding-dingeringeding!
Wa-pa-pa-pa-pa-pa-pow!
Hatee-hatee-hatee-ho!
Joff-tchoff-tchoffo-tchoffo-tchoff!
```

Another type of closure is with procs. A function can be created like a variable with Proc.new and the function to be stored can be given in a block. In the above example, a function that takes a variable and embeds it into a string is stored in the variable whatDoesTheFoxSay. This stored function can be called with ".call" and passing the argument.

```
favDogBreeds = -> (*breeds) {puts "My fav dog breeds are #{breeds}"}
favDogBreeds.call("Labrador", "Husky", "Pinscher", "Italian Greyhound")

Output:
My fav dog breeds are ["Labrador", "Husky", "Pinscher", "Italian Greyhound"]
```

Lambda in Ruby is similar to callback functions in JavaScript and this is also very similar to the procs structure. Again a function can be stored inside a variable and it can be called from the variable. In the above example, an args array is the parameter of the stored function and this function prints this array. The purpose of this function is to print favorite dog breeds! It is called with call method and it printed my favorite dog breeds!

8. Evaluation of Language

Ruby has a simple syntax overall when it comes to subprograms. I really liked the print function, it is original :). Defining functions is easy; however, I forgot to add the end keyword several times since I am used to using curly brackets. This is a minus for me since it decreases fluency in writing. It is also hard to match the end keywords with functions when there are too many nested subprograms, this decreases readability for me. However, defining variables is easy since we do not need to specify types for variables and it makes this language extremely writable in my opinion. In terms of return values, it is nice to have multiple return values, and it certainly increases the writability of the program but it decreases reliability because you can end up assigning an array to a variable instead of a single integer if the function returns multiple values but you did not know it. Moreover, it also reduces the writability to have return values even without the return keyword since sometimes I want to have void functions. As far as I understand, to achieve void functions we should provide an empty return, but this is something I am not used to doing. Moreover, closures are not easy to write or read in my opinion. For example, syntax is like calling a function from a number in "3.times" and the parameters in blocks are given between "|". These are not so understandable at first sight, and the code becomes hard to follow which decreases readability. Having default parameters and keyword variables are increasing the

writability by providing flexibility. Moreover, being able to pass as many parameters as we wish with args* increases the writability by also providing flexibility. Not having method overloading as in Java is decreasing writability but it increases readability since there is only one valid function with a specific name at a time. Moreover, the nested subprograms are not very useful in terms of writability since it is not possible to access the wrapper function's variables if we do not define the wrapper function's variables global. Also, the subprogram cannot be used as closures, only blocks can be used for this purpose. Overall, Ruby has a syntax similar to Python which makes it readable to some extent but it is not the most readable language since closures are hard to follow, and it is not also the most writable language in terms of subprograms since subprograms are not behaving as static children or parents of each other.

9. My Learning Strategy

As in the previous two homework, I skimmed through my notes to remember the concepts of closure, parameter passing methods, and keyword parameters. After doing that I searched our textbook's PDF file to find information about the Ruby language relevant to design issues given in homework [1]. It was a little difficult for me to find examples for these design issues since there are no Ruby examples in Altay Hoca's example codes. Nevertheless, I managed to find examples related to these design issues in official documentation and on online forums such as Stackoverflow [2]. After remembering the design issues that confuse me, and after seeing enough examples in this language, I felt ready to start this homework. For this homework, I did not use an online compiler. As Altay Hoca has suggested, I have downloaded the Ruby compiler to my computer and downloaded the Ruby extension for Visual Studio Code [3], [5]. I started coding with return values since I thought it was the easiest design issue. However, I was surprised to see that it was possible to return multiple values from functions, and also the language returns the last value automatically. For the return values part, I got help from a Youtube video about Ruby Language, a blog about Ruby for Beginners, and two blogs on Medium [3], [4], [11], [12]. After finishing this part, I continued with nested subprogram definitions, it was straightforward, and it was possible to call nested subprograms from outside levels. I did not look up sources in this part, and I have completed it by experimenting. After that, I wrote programs for the scope of local variables. This part was less surprising and the local variables inside a subprogram were only visible in that subprogram as in PHP. For this part, I have read the official documentation of Ruby [2]. In Stackoverflow and GeekforGeeks, there were entries saying that there is no method overloading but the textbook was saying there is method overloading in Ruby [1], [9]. I have concluded that it has method overloading and has no method overloading at the same time. I have added a small program showing that method overloading exists to some extent but two methods cannot be used at the same time. After completing this part, I continued with keyword and default parameters, the book was saying Ruby does not have keyword arguments but after searching for that I have seen that keyword arguments were introduced in Ruby 2 and Ruby 3 also has them [1], [6], [7], [8]. I also read two blog entries for this part in

Medium. In the final part, I wrote programs for closures. This was the most confusing part of the homework for me since I did not remember the exact definition of the closure and there were three types of closures in Ruby with a weird syntax [2]. Geeksforgeeks helped me understand closures in Ruby, and I managed to complete this part [13]. Then I added code snippets and outputs to the report and added explanations for all snippets. Overall, it was a good experience for me. Ruby has a simple syntax and it was good to see I manage to have a basic understanding of a language that is completely new to me by reading docs and blogs.

10. References

[1] Robert W. Sebesta, *Concepts of Programming Languages*. Pearson, 2016.

[2] “Programming Ruby”, ruby-doc. Available: http://ruby-doc.com/docs/ProgrammingRuby/ [Accessed on: May 4, 2022].

[3] “Ruby Programming Language - Full Course”, FreeCodeCamp. Available: https://www.youtube.com/watch?v=t_ispmWmdjY&t=5843s. [Accessed on: May 4, 2022].

[4] “Return Values”, Ruby for Beginners. Available: http://ruby-for-beginners.rubymonstas.org/writing_methods/return_values.html. [Accessed on: May 4, 2022].

[5] “Visual Studio Code”. Available: https://code.visualstudio.com. [Accessed on: May 4, 2022].

[6] Orawo, G., “Ruby Required, Default and Optional Parameters”, Medium. Available: https://medium.com/podiihq/ruby-parameters-c178fdcd1f4e. [Accessed on: May 4, 2022].

[7] Sologub, A., “Parameters with default values in ruby”, Medium. Available: https://medium.com/@sologoubalex/parameters-with-default-values-in-ruby-74cd0e830681. [Accessed on: May 4, 2022].

[8] “Separation of positional and keyword arguments in Ruby 3.0”, Ruby-lang. Available: https://www.ruby-lang.org/en/news/2019/12/12/separation-of-positional-and-keyword-arguments-in-ruby-3-0/. [Accessed on: May 4, 2022].

[9] “Method Overloading In Ruby”, GeeksforGeeks. Available: https://www.geeksforgeeks.org/method-overloading-in-ruby/. [Accessed on: May 4, 2022].

[10] “Local variables”, Ruby-doc. Available: https://ruby-doc.org/docs/ruby-doc-bundle/UsersGuide/rg/localvars.html. [Accessed on: May 4, 2022].

[11] “The return Keyword in Ruby”, Medium. Available: https://medium.com/rubycademy/the-return-keyword-in-ruby-df0a7f578fcb. [Accessed on: May 4, 2022].

[12] Sologub, A., G. “An introduction to method return value in ruby”, Medium. Available: https://medium.com/@sologoubalex/an-introduction-to-method-return-value-in-ruby-5456cfdd8b90. [Accessed on: May 4, 2022].

[13] “Closures in Ruby”, GeeksforGeeks. Available: https://www.geeksforgeeks.org/closures-in-ruby/#:~:text=In%20Ruby%2C%20closure%20is%20a,any%20function%20as%20an%20argument.. [Accessed on: May 4, 2022].