2021–2022 SPRING SEMESTER

CS315 – HW3

Subprograms in Ruby

NAME: Abdullah Riaz
ID: 22001296
COURSE: CS315 - Programming Languages
SECTION: 01
DATE: 06/05/2022

Subprogram overloading

- Code

```ruby
def ovrld_exm (p1)

  puts "This subprogram, ovrld_exm, takes #{p1} parameters"

end

def ovrld_exm (p1, p2)

  puts "This subprogram, ovrld_exm, takes #{p2+p1} parameters"

end


#ovrld_exm 1  //This will not compile
ovrld_exm 1,1
```

- Output

```
This subprogram, ovrld_exm, takes 2 parameters
```

- Discussion

  Ruby, unlike other languages, does not support subprogram overloading. In the code above, while we do declare and overload a subprogram with two different parameter profiles, the code will not compile if the subprogram is called with a single parameter. The language decides which subprogram variant to use based on which is declared most recently. For example, suppose the subprogram with two parameters was declared first, and the one with one parameter was declared later. In that case, the program would not compile unless the subprogram is called with a single parameter.

Return values

- Code

```ruby
def add_five_prog(val)  #takes any value and adds 5 to it
  val + 5
end

p add_five_prog(5)
```

- Output

```
10
```

- Discussion

  In Ruby, a subprogram always returns an object. It returns the value from the last statement in the subprogram. In the code above, the last statement calculated a value of 10; hence, it was in the output.

- Code

```ruby
def add_five_prog(val)  #takes any value and adds 5 to it
  val + 5
  puts "15"
end

p add_five_prog(5)
```

- Output

```
15
nil
```

- Discussion Cont.

  In this case, the last evaluated statement was a *'puts'* (a statement used to print). This caused the return value to be a "nil" object which is the default return value for ruby subprograms if one is not specified. The *'return'* keyword can be used as such

  ```
  return val +  5,
  ```

  which will return the evaluated *'val'*. However, it would also finish evaluating the subprogram at that point and ignore any other statements after it. This is why the statement that calculates the return value is usually the last in a Ruby subprogram and why the return keyword is not used a lot in Ruby.

Nested Subprogram Definitions

- Code

```
def add5(x)    #subprogram  thats adds 5 to value
  def add_prog(y)  #nested subprogram that does the addition
    y + 5
  end
puts add_prog x    #print the return value
end

add5 10
```

- Output

```
15
```

- Discussion

  Ruby supports definitions of subprograms within other subprograms. In the code example above, we defined a subprogram within another subprogram. The nested subprogram adds 5 to a value passed to it, whereas the top subprogram just passes the value(from the main program) to it. It works as can be expected as seen from the output.

Scope of local variables

- Code

```
def add5(x)    #subprogram  thats adds 5 to value
  z = 6
  def add_prog(y)  #nested subprogram that does the addition
    y + 5
    #z + 5    #accessing parent subprogram variable give an error
  end

puts add_prog x    #print the return value
end


add5 10
```

- Output

```
15
```

- Discussion

  In Ruby subprograms, the scope of variable is limited to the definition block of the subprogram. A subprogram can only access variables defined inside it(global variables are an exception), and any attempt otherwise leads to a compile error. In the code

above, this statement `z + 5` gives a compilation error when run. This is because the variable ‘z’ is defined in the parent subprogram and not inside the `add_prog` subprogram where the statement is being evaluated.

Parameter passing methods

- Code

```
def pass_case (x)
    x += 1
    puts "Value of x inside subprogram #{x}"
end


x = 5
pass_case x
puts "Value of x after subprogram: #{x}"
```

- Output

```
Value of x inside subprogram 6
Value of x after subprogram: 5
```

- Discussion

In Ruby, the parameter passing technique is pass-by-assignment. It essentially means that the parameter that is passed to the subprogram is not modified. The subprogram creates a local variable with a value copied from the argument in the main program and modifies it, and returns that value as well.
From the output, we can see the value of our variable x in the main program remained the same despite being modified in the subprogram.

Keyword and default parameters

- Code

```
def defult_exm (p1 = 5, p2 = 5)

  puts p1 + p2

end

 defult_exm  # without any params
 defult_exm 1 #single params
 defult_exm 1, 10 #all params
```

- Output

```
10
6
11
```

- Discussion

  In Ruby, keyword parameter are not supported. Default parameters are, however, supported. In the code example above, we define a program with default parameters and call it in the main program with a different number of arguments to test it.
  No arguments are passed in the first call, so it adds the default values. In the second call, it takes one argument and adds it with the second default variable. The third call takes both arguments, uses them for addition, and completely ignores the default values.

Closures

- Code

```
def block_prog
    yield
    puts "This is inside the block subprogram"
    yield
end

block_prog { puts "This statement is from the caller"}
```

- Output

```
This statement is from caller
doing subprogram stuff
This statement is from caller
```

- Discussion

  In Ruby, there is support for closures. One of the closures used in subprograms is Blocks. The code above is an example of a block. Whenever a *yield* keyword is used in a subprogram, It runs the statement inside the curly brackets from the statement inside the main program, which calls the subprogram. It's akin to do while loops of other programming languages like C++ where the *yield* keyword is the condition to execute the statement inside the *do* bracket. The output shows this implementation as whenever *yield* is used, it outputs the curly bracket statement

Learning Strategy

I started with the coursebook “Concepts of Programming Languages” chapter 9 regarding the subprograms. After understanding the terminology and basic idea behind it, I went to the online compiler and started writing code to test the design issues. I would frequently visit the official documentation whenever I would stumble on syntax issues or if the output wasn't as I expected. For example, I wrongly assumed the passing methods of parameters, and the result wasn't as I expected. After looking at the docs, I understood I was going around the wrong way. This is how I went around doing the assignment.

References:

Replit. (n.d.). *Ruby online compiler & interpreter*. replit. Retrieved May 6, 2022, from https://replit.com/languages/ruby

Ruby. Ruby Programming Language. (n.d.). Retrieved May 6, 2022, from https://www.ruby-lang.org/en/