2022 Spring

CS315

Homework Assignment 3

Subprograms in Ruby

Name: Ahmet Faruk

Surname: Ulutaş

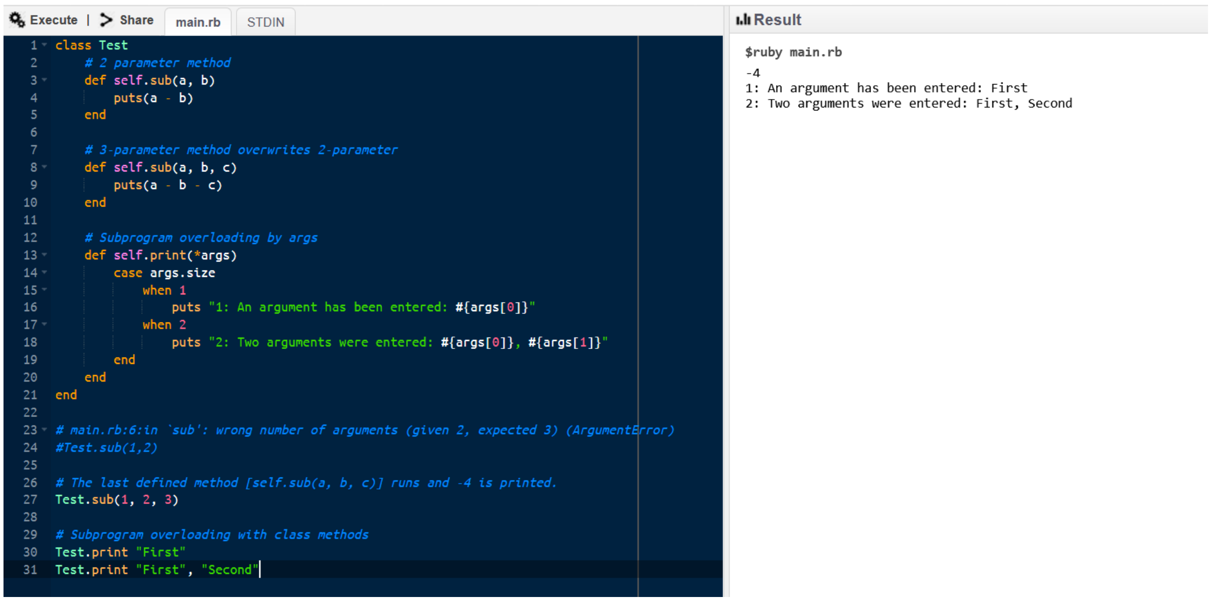
Section: 1

ID: 21803717

TA Name: Irmak Türköz

Subprogram Overloading

Overloading is the binding of methods statically at compile time. However, Ruby is a language that provides dynamic binding. Therefore, only the last subprogram defined is valid. However, partial overloading can be achieved by making the argument when-case in methods within the class.

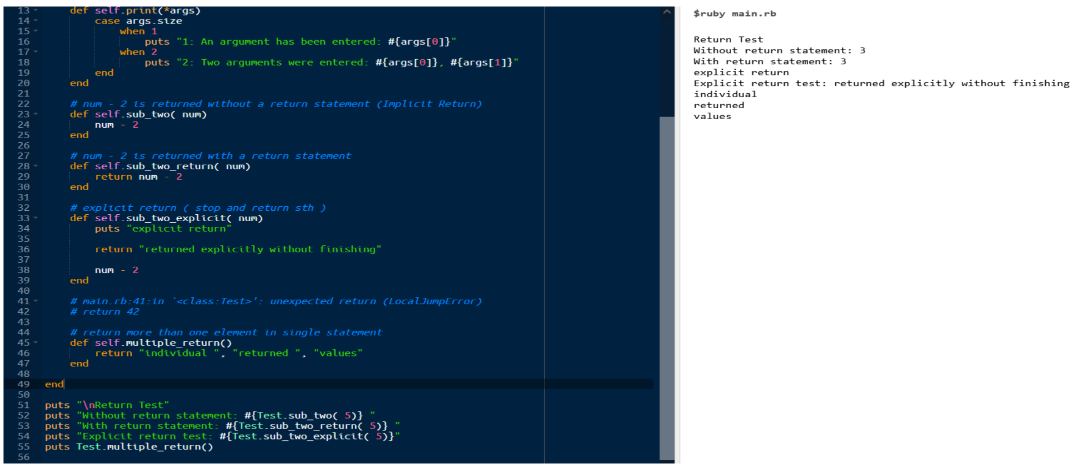


```ruby
class Test
    # 2 parameter method
    def self.sub(a, b)
        puts(a - b)
    end

    # 3-parameter method overwrites 2-parameter
    def self.sub(a, b, c)
        puts(a - b - c)
    end

    # Subprogram overloading by args
    def self.print(*args)
        case args.size
            when 1
                puts "1: An argument has been entered: #{args[0]}"
            when 2
                puts "2: Two arguments were entered: #{args[0]}, #{args[1]}"
        end
    end
end

# main.rb:6:in `sub': wrong number of arguments (given 2, expected 3) (ArgumentError)
#Test.sub(1,2)

# The last defined method [self.sub(a, b, c)] runs and -4 is printed.
Test.sub(1, 2, 3)

# Subprogram overloading with class methods
Test.print "First"
Test.print "First", "Second"
```

```
$ruby main.rb
-4
1: An argument has been entered: First
2: Two arguments were entered: First, Second
```

Return Values

In Ruby, each method can always return only one object. This object can be nil. If more than one value or variable is wanted to be returned, an array can be used. In addition, the return statement does not have to be written. If no return statement is written, the result of the last evaluated statement is returned. The function can be terminated by making an explicit return before the program definition is completed. Also, if it is returned outside of the function, an unexpected return error is received.

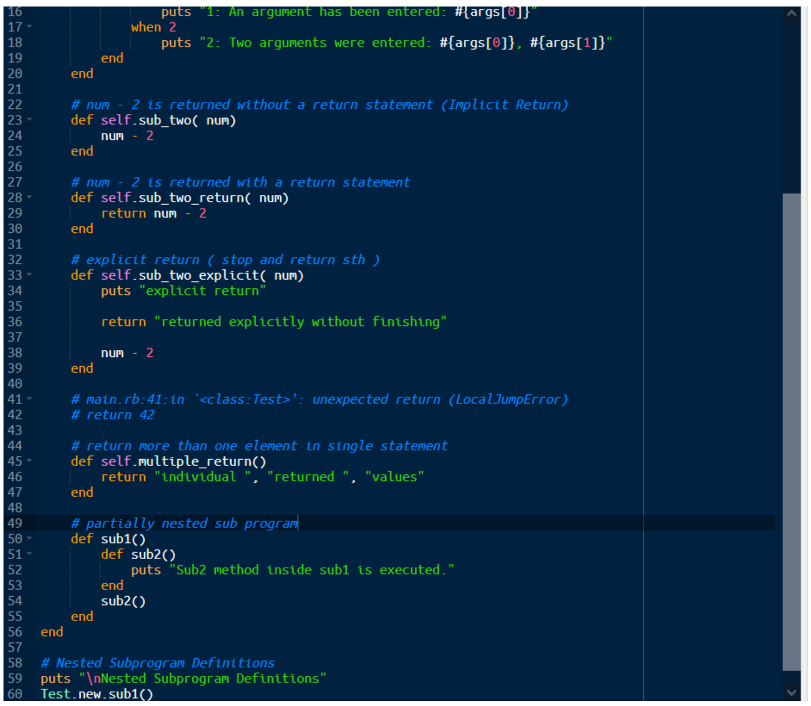


```ruby
    def self.print(*args)
        case args.size
            when 1
                puts "1: An argument has been entered: #{args[0]}"
            when 2
                puts "2: Two arguments were entered: #{args[0]}, #{args[1]}"
        end
    end

    # num - 2 is returned without a return statement (Implicit Return)
    def self.sub_two( num)
        num - 2
    end

    # num - 2 is returned with a return statement
    def self.sub_two_return( num)
        return num - 2
    end

    # explicit return ( stop and return sth )
    def self.sub_two_explicit( num)
        puts "explicit return"

        return "returned explicitly without finishing"

        num - 2
    end

    # main.rb:41:in `<class:Test>': unexpected return (LocalJumpError)
    # return 42

    # return more than one element in single statement
    def self.multiple_return()
        return "individual ", "returned ", "values"
    end

end

puts "\nReturn Test"
puts "Without return statement: #{Test.sub_two( 5)} "
puts "With return statement: #{Test.sub_two_return( 5)} "
puts "Explicit return test: #{Test.sub_two_explicit( 5)}"
puts Test.multiple_return()
```

```
$ruby main.rb

Return Test
Without return statement: 3
With return statement: 3
explicit return
Explicit return test: returned explicitly without finishing
individual
returned
values
```

Nested Subprogram Definitions

Since Ruby uses dynamic binding, it does not allow nested subprograms. However, it can still be partially implemented by writing a nested method and calling the method.

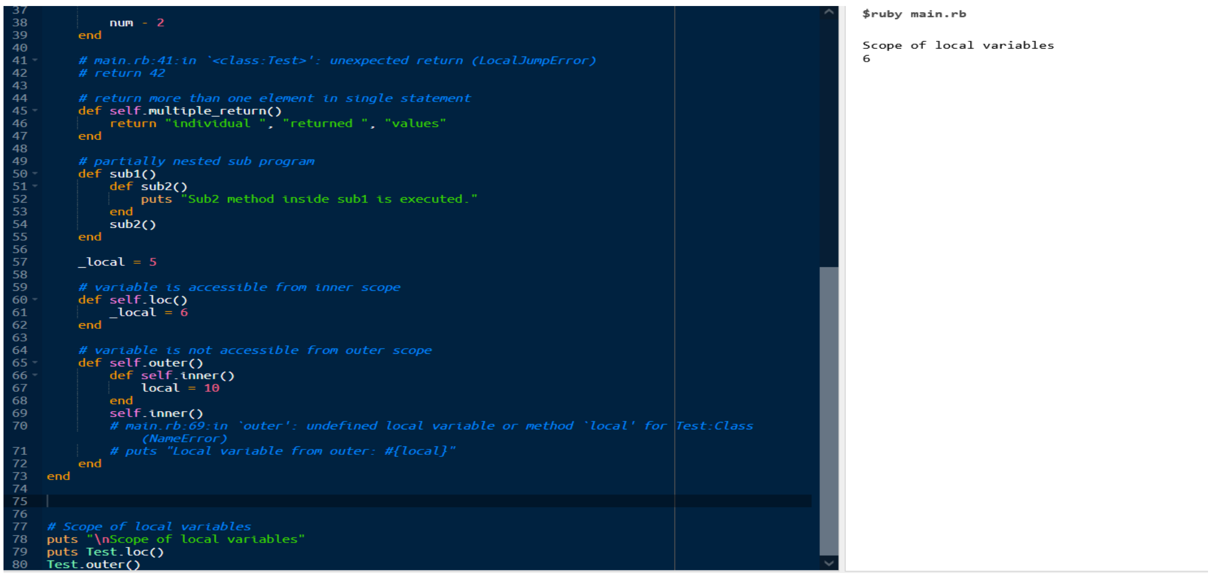


```ruby
                puts "1: An argument has been entered: #{args[0]}"
            when 2
                puts "2: Two arguments were entered: #{args[0]}, #{args[1]}"
        end
    end

    # num - 2 is returned without a return statement (Implicit Return)
    def self.sub_two( num)
        num - 2
    end

    # num - 2 is returned with a return statement
    def self.sub_two_return( num)
        return num - 2
    end

    # explicit return ( stop and return sth )
    def self.sub_two_explicit( num)
        puts "explicit return"

        return "returned explicitly without finishing"

        num - 2
    end

    # main.rb:41:in `<class:Test>': unexpected return (LocalJumpError)
    # return 42

    # return more than one element in single statement
    def self.multiple_return()
        return "individual ", "returned ", "values"
    end

    # partially nested sub program
    def sub1()
        def sub2()
            puts "Sub2 method inside sub1 is executed."
        end
        sub2()
    end
end

# Nested Subprogram Definitions
puts "\nNested Subprogram Definitions"
Test.new.sub1()
```

```
$ruby main.rb

Nested Subprogram Definitions
Sub2 method inside sub1 is executed.
```

Scope of Local Variables

Local variables cannot be accessed outside the scope in which they are defined. Variable names must be defined with underscores or lowercase letters.

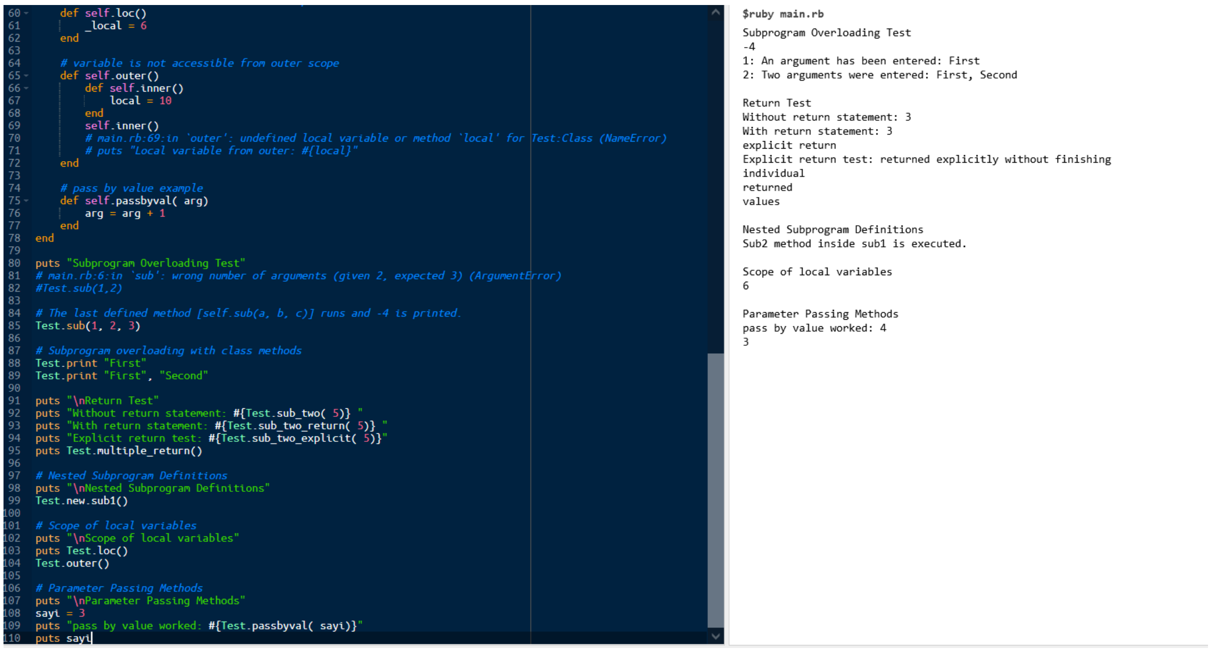


```ruby

        num - 2
    end

    # main.rb:41:in `<class:Test>': unexpected return (LocalJumpError)
    # return 42

    # return more than one element in single statement
    def self.multiple_return()
        return "individual ", "returned ", "values"
    end

    # partially nested sub program
    def sub1()
        def sub2()
            puts "Sub2 method inside sub1 is executed."
        end
        sub2()
    end

    _local = 5

    # variable is accessible from inner scope
    def self.loc()
        _local = 6
    end

    # variable is not accessible from outer scope
    def self.outer()
        def self.inner()
            local = 10
        end
        self.inner()
        # main.rb:69:in `outer': undefined local variable or method `local' for Test:Class (NameError)
        # puts "Local variable from outer: #{local}"
    end
end



# Scope of local variables
puts "\nScope of local variables"
puts Test.loc()
Test.outer()
```

```
$ruby main.rb

Scope of local variables
6
```

Parameter Passing Methods

In pass-by-value, the value is passed to the method and changes in the method do not affect the actual variable. In pass-by-reference, the reference of the variable is passed to the method and the actual variable is affected by all changes. The Ruby language works as pass-by-value.

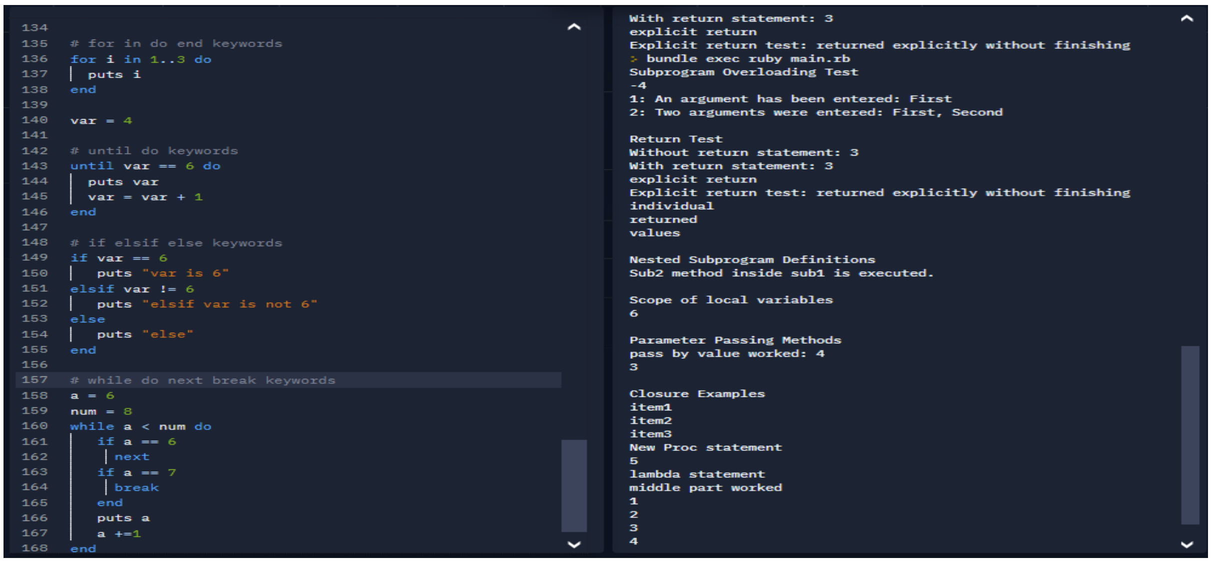


```ruby
    def self.loc()
        _local = 6
    end

    # variable is not accessible from outer scope
    def self.outer()
        def self.inner()
            local = 10
        end
        self.inner()
        # main.rb:69:in `outer': undefined local variable or method `local' for Test:Class (NameError)
        # puts "Local variable from outer: #{local}"
    end

    # pass by value example
    def self.passbyval( arg)
        arg = arg + 1
    end
end

puts "Subprogram Overloading Test"
# main.rb:6:in `sub': wrong number of arguments (given 2, expected 3) (ArgumentError)
#Test.sub(1,2)

# The last defined method [self.sub(a, b, c)] runs and -4 is printed.
Test.sub(1, 2, 3)

# Subprogram overloading with class methods
Test.print "First"
Test.print "First", "Second"

puts "\nReturn Test"
puts "Without return statement: #{Test.sub_two( 5)} "
puts "With return statement: #{Test.sub_two_return( 5)} "
puts "Explicit return test: #{Test.sub_two_explicit( 5)}"
puts Test.multiple_return()

# Nested Subprogram Definitions
puts "\nNested Subprogram Definitions"
Test.new.sub1()

# Scope of local variables
puts "\nScope of local variables"
puts Test.loc()
Test.outer()

# Parameter Passing Methods
puts "\nParameter Passing Methods"
sayi = 3
puts "pass by value worked: #{Test.passbyval( sayi)}"
puts sayi
```

```
$ruby main.rb
Subprogram Overloading Test
-4
1: An argument has been entered: First
2: Two arguments were entered: First, Second

Return Test
Without return statement: 3
With return statement: 3
explicit return
Explicit return test: returned explicitly without finishing
individual
returned
values

Nested Subprogram Definitions
Sub2 method inside sub1 is executed.

Scope of local variables
6

Parameter Passing Methods
pass by value worked: 4
3
```

Keyword and Default Parameters

There are 41 public keywords in Ruby. The most used ones can be listed as follows: begin, end, and, break, case, class, def, do, else, elsif, for, if, in, next, nil, not, or, redo, rescue, retry, return, self, then, unless, until, when, while, yield.

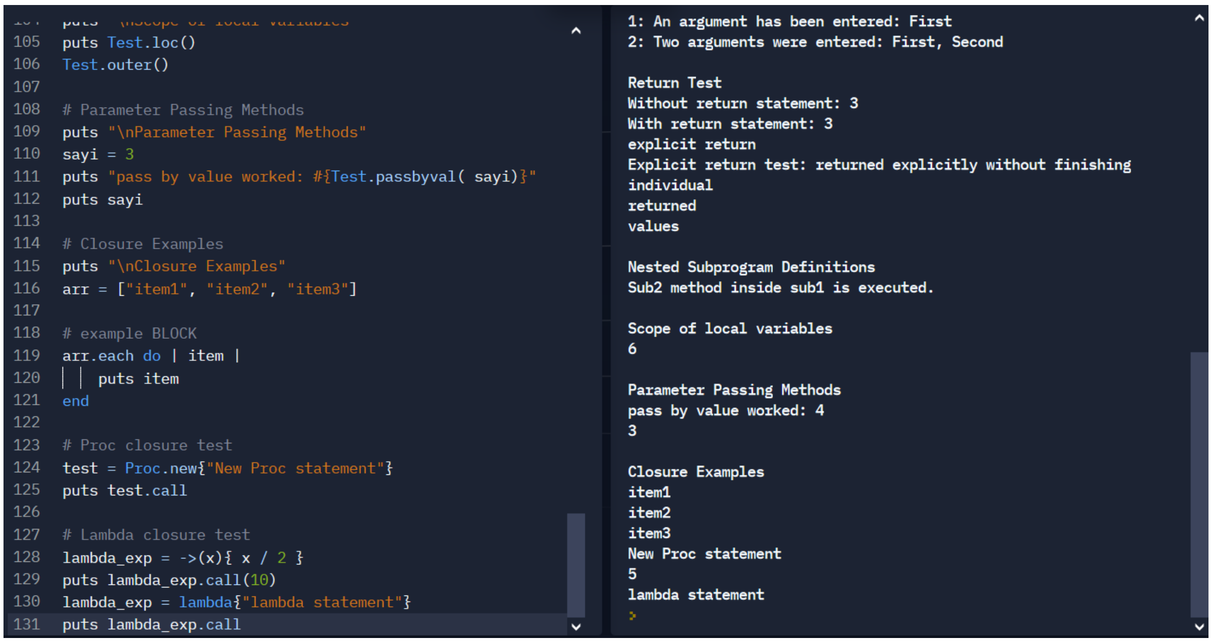


```ruby

# for in do end keywords
for i in 1..3 do
  puts i
end

var = 4

# until do keywords
until var == 6 do
  puts var
  var = var + 1
end

# if elsif else keywords
if var == 6
   puts "var is 6"
elsif var != 6
   puts "elsif var is not 6"
else
   puts "else"
end

# while do next break keywords
a = 6
num = 8
while a < num do
   if a == 6
     next
   if a == 7
     break
   end
   puts a
   a +=1
end
```

```
With return statement: 3
explicit return
Explicit return test: returned explicitly without finishing
> bundle exec ruby main.rb
Subprogram Overloading Test
-4
1: An argument has been entered: First
2: Two arguments were entered: First, Second

Return Test
Without return statement: 3
With return statement: 3
explicit return
Explicit return test: returned explicitly without finishing
individual
returned
values

Nested Subprogram Definitions
Sub2 method inside sub1 is executed.

Scope of local variables
6

Parameter Passing Methods
pass by value worked: 4
3

Closure Examples
item1
item2
item3
New Proc statement
5
lambda statement
middle part worked
1
2
3
4
```

Closures

In Ruby, a closure represents the transaction block depending on where it is called, and there are three types of closures: Blocks, Procs, Lambda.



```ruby
puts Test.loc()
Test.outer()

# Parameter Passing Methods
puts "\nParameter Passing Methods"
sayi = 3
puts "pass by value worked: #{Test.passbyval( sayi)}"
puts sayi

# Closure Examples
puts "\nClosure Examples"
arr = ["item1", "item2", "item3"]

# example BLOCK
arr.each do | item |
    puts item
end

# Proc closure test
test = Proc.new{"New Proc statement"}
puts test.call

# Lambda closure test
lambda_exp = ->(x){ x / 2 }
puts lambda_exp.call(10)
lambda_exp = lambda{"lambda statement"}
puts lambda_exp.call
```

```
1: An argument has been entered: First
2: Two arguments were entered: First, Second

Return Test
Without return statement: 3
With return statement: 3
explicit return
Explicit return test: returned explicitly without finishing
individual
returned
values

Nested Subprogram Definitions
Sub2 method inside sub1 is executed.

Scope of local variables
6

Parameter Passing Methods
pass by value worked: 4
3

Closure Examples
item1
item2
item3
New Proc statement
5
lambda statement
>
```

Evaluation of Ruby in Terms of Readability and Writability of Subprogram Syntax

I think the Ruby language is pretty good in terms of readability. The keywords used for methods and blocks are familiar and self-explanatory. Features such as the fact that you do not need to define anything extra, for example, that you can directly write puts and output, greatly increase readability. The same situation increases the writability. Because you don't need class and static main definitions like in Java or C-based languages. You can write the program you want to write in a very short way. In addition, the breadth of keywords provides good flexibility in writing. However, I think that cases like pass-by-reference are not direct, which reduces writability. It's not always good for people to use procs or lambda expressions. However, it can be said that its writability is high because it is very easy to write code.

My Learning Strategy

While applying my learning strategy, I first did the necessary research on the topics that I would write in Ruby. Then I found an online compiler that works fine and I started to write the codes there. While I was writing my codes, I found it a bit strange that there were only dynamic uses in the ruby language. However, I am surprised that this makes writability considerably easier and simplifies the language. Although I was worried that only one object should always be returned in the return statement, I saw that it is not lacking in other languages. I thought it was a bad feature that you couldn't define nested programs. However, since dynamic binding is used, I considered this situation normal. I am surprised that the scope of the variable cannot be accessed externally and can only be accessed internally. Considering all these features together, Ruby is written as a very plain and simple language. Then my online compiler was broken and I had to switch to a new compiler. I switched from TutorialsPoint to ReplIt. I think that this simplicity makes it easier to write Proc and Lambda statements that can be easily identified among them. While completing this assignment, I tested and observed the relevant scenarios in all references in my sources as code. As a result, I added a small summary section and wrote my own codes based on this summary. I found the Ruby language quite suitable for small and fast programming uses. However, I think it has a limitation that comes from simplicity, as it cannot be used in large projects on an application development basis.

References

https://www.geeksforgeeks.org/method-overloading-in-ruby/

http://ruby-for-beginners.rubymonstas.org/writing_methods/return_values.html

https://medium.com/rubycademy/the-return-keyword-in-ruby-df0a7f578fcb

https://dev.to/viricruz/how-to-return-multiple-values-in-ruby-1cb

https://stackoverflow.com/questions/4864191/is-it-possible-to-have-methods-inside-methods

https://www.techotopia.com/index.php/Ruby_Variable_Scope

https://www.geeksforgeeks.org/ruby-keywords/

https://learn.co/lessons/methods-default-arguments

https://mixandgo.com/learn/ruby/pass-by-reference-or-value

https://www.geeksforgeeks.org/closures-in-ruby/

https://www.tutorialspoint.com/execute_ruby_online.php

https://replit.com/languages/ruby