

SWINBURNE UNIVERSITY OF TECHNOLOGY

COS20007 OBJECT ORIENTED PROGRAMMING

---

# Preparing for Object Oriented Programming

---

PDF generated at 20:32 on Sunday 6<sup>th</sup> August, 2023

# 1.1P: Preparing for OOP – Answer Sheet

1. Explain the following terminal instructions:
  - a. `cd`: Change directory, used to navigate to a different folder
  - b. `ls`: List files and directories in the current folder.
  - c. `pwd`: Print the current working directory
2. Consider the following kinds of information, and suggest the most appropriate data type to store or represent each:

Information	Suggested Data Type
A person's name	String
A person's age in years	Integer
A phone number	String
A temperature in Celsius	Float
The average age of a group of people	Float
Whether a person has eaten lunch	Integer

3. Aside from the examples already provided in question 2, come up with an example of information that could be stored as:

Data type	Suggested Information
String	Name = "Quang Vinh Le"
Integer	Age = 19
Float	GPA = 3.0
Boolean	IsStudent = true

4. Fill out the last two columns of the following table, evaluating the value of each expression and identifying the data type the value is most likely to be:

Expression	Given	Value	Data Type
6		6	Integer
True		True	Boolean

a	a = 2.5	2.5	Float
1 + 2 * 3		7	Integer
a and False	a = True	False	Boolean
a or False	a = True	True	Boolean
a + b	a = 1 b = 2	3	Integer
2 * a	a = 3	6	Integer
a * 2 + b	a = 2.5 b = 2	7	Float
a + 2 * b	a = 2.5 b = 2	6.5	Float
(a + b) * c	a = 1 b = 1 c = 5	10	Integer
"Fred" + " Smith"		Fred Smith	String
a + " Smith"	a = "Wilma"	Wilma Smith	String

5. Using an example, explain the difference between **declaring** and **initialising** a variable.

The difference between the two is:

- Declaring a variable means defining its existence and data type without assigning it a specific value. Example: `age = int (python)`
- Initializing a variable means assigning it a value for the first time after it has been declared. Example: `age = 19 (python)`

6. Explain the term **parameter**. Write some code that demonstrates a simple use of a parameter. You should show a procedure or function that uses a parameter, and how you would call that procedure or function.

- A parameter is a variable or placeholder in a function or procedure that allows you to pass values into the function when it is called.  
*<insert a screenshot of your code here>*

```
def greet_person(name)
  puts "Hello, #{name}! How are you today?"
end

greet_person("Vinh")
```

- In this example, I defined a function called "greet\_person" that takes one parameter named "name." The function returns a greeting message with the provided name. Then I call the function and pass the value "Vinh" as the argument for the "name" parameter. Finally, we print the greeting message, resulting in the output: "Hello, Vinh! How are you today?"

7. Using an example, describe the term **scope** as it is used in procedural programming (not in business or project management). Make sure you explain the different kinds of scope.

- Scope is a concept in procedural programming that defines the visibility and accessibility of variables within different parts of the code. It determines where a variable can be accessed and used during the execution of the program. There are two main kinds of scope in procedural programming:
  1. Global Scope:
    - Variables declared in the global scope are accessible throughout the entire program, meaning they can be accessed from any part of the code, including functions and blocks.
    - Global variables are declared outside of any function or block, making them accessible from any part of the program.
    - Global variables retain their value throughout the program's execution, and any changes made to them in one part of the program are visible in other parts.

Example:

```
COS20007 > testrb
1  # Global variable declared outside any function
2  $global_var = 10
3
4  # Function that uses the global variable
5  def print_global_var
6    puts "Global variable inside the function: #{$global_var}"
7  end
8
9  # Function that modifies the global variable
10 def modify_global_var
11   $global_var = 20
12 end
13
14 # Call the functions
15 print_global_var # Output: Global variable inside the function: 10
16 modify_global_var
17 print_global_var # Output: Global variable inside the function: 20
18
```

- In this example, `$global_var` is a global variable accessible both inside and outside the functions. The function `print_global_var` accesses and prints the value of the global variable. When we call `modify_global_var`, it modifies the value of `$global_var`, and when we call `print_global_var` again, the updated value is printed.

## 2. Local Scope:

-Variables declared inside a function have local scope, meaning they can only be accessed within that specific function.

-Local variables are created when the function is called and destroyed when the function exits. They do not persist beyond the function's execution.

-Local variables may have the same name as global variables, but they are distinct and have no impact on each other.

Example:

```
COS20007 > test.rb
1  # Global variable
2  $global_var = 10
3
4  # Function with local variable
5  def some_function
6    local_var = 5
7    puts "Local variable inside the function: #{local_var}"
8    puts "Global variable inside the function: #{global_var}"
9  end
10
11 # Call the function
12 some_function
13 # Output: Local variable inside the function: 5
14 #          Global variable inside the function: 10
```

- In this example, `local_var` is a local variable, and it can only be accessed within the `some_function`. Attempting to access `local_var` outside the function will result in an error, as it is confined to the local scope of the function.

8. In a procedural style, in any language you like, write a function called `Average`, which accepts an array of integers and returns the average of those integers. Do not use any libraries for calculating the average. You must demonstrate appropriate use of parameters, returning and assigning values, and use of a loop. Note — just write the function at this point, we'll use it in the next task. You shouldn't have a complete program or even code that outputs anything yet at the end of this question.

<insert a screenshot of your code here>

```
COS20007 > test.rb
1  def average(arr)
2      # Check if the array is empty to avoid division by zero
3      return 0 if arr.empty?
4
5      sum = 0
6      num_elements = arr.length
7
8      # Use a while loop to calculate the sum of all elements in the array
9      i = 0
10     while i < num_elements
11         sum += arr[i]
12         i += 1
13     end
14
15     # Calculate the average by dividing the sum by the number of elements
16     average = sum.to_f / num_elements
17
18     return average
19 end
```

- In this implementation, we define the 'average' function that takes an array of integers as a parameter. The function first checks if the array is empty to avoid division by zero and returns 0 in such cases.
- We then initialize a variable sum to keep track of the sum of all elements in the array, and another variable 'num\_elements' to store the length of the array.
- Next, we use a 'while' loop to iterate through the array and calculate the sum of all elements. The variable 'i' serves as the loop counter, and it is incremented at each iteration to access the next element in the array until we reach the end.
- Finally, we calculate the average by dividing the sum by the number of elements in the array and return the result.

9. In the same language, write the code you would need to call that function and print out the result.

<insert a screenshot of your code here>

```
20
21  numbers = [10, 20, 30, 40, 50]
22  result = average(numbers)
23  puts "Average: #{result}"
24
```

- An example usage of the 'average' function is provided. The 'numbers' array contains [10, 20, 30, 40, 50], and the function is called with this array. The result of the average calculation is stored in the variable 'result', and it is printed to the console using puts. Output: "Average: 30". This is the average

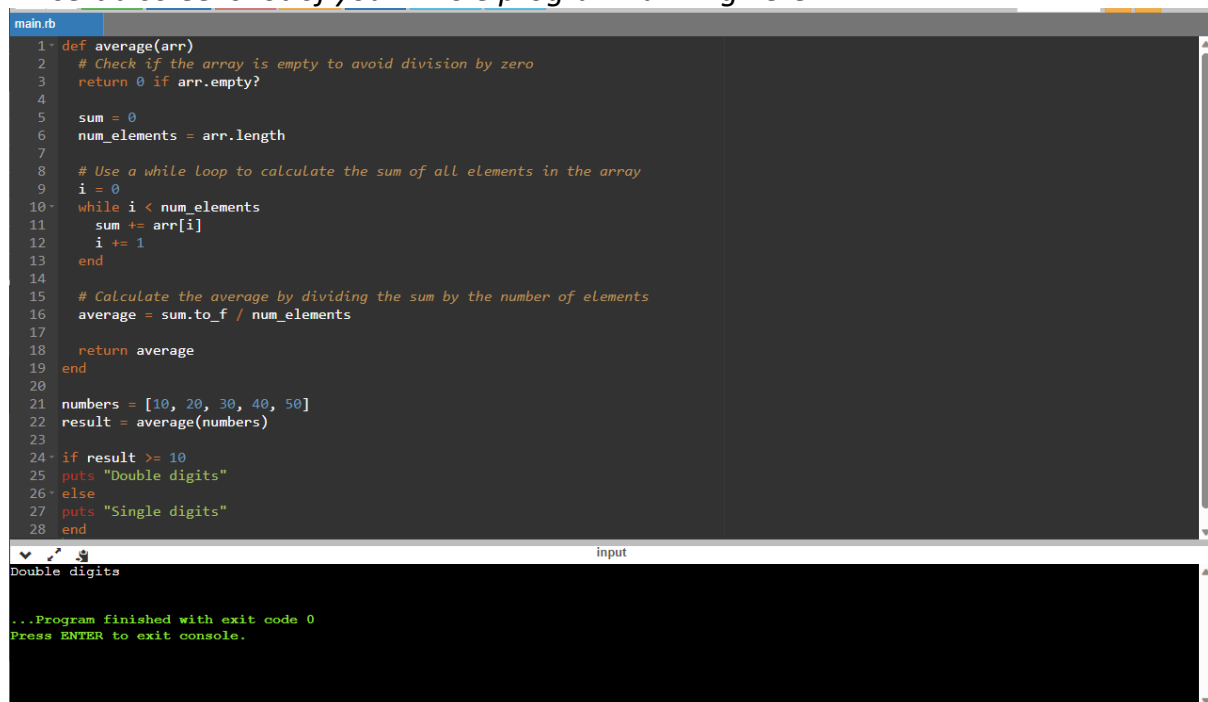
of the elements in the numbers array [10, 20, 30, 40, 50].

10. To the code from 9, add code to print the message "Double digits" if the average is above or equal to 10. Otherwise, print the message "Single digits". Provide a screenshot of your program running.

<insert a screenshot of your code here>

```
20
21   numbers = [10, 20, 30, 40, 50]
22   result = average(numbers)
23
24   if result >= 10
25     puts "Double digits"
26   else
27     puts "Single digits"
28   end
29
```

<insert a screenshot of your whole program running here>



The screenshot shows a Ruby IDE with a file named 'main.rb'. The code defines an 'average' function that calculates the average of an array. It then uses this function on the array [10, 20, 30, 40, 50] and prints 'Double digits' because the average (30) is greater than or equal to 10. The output window at the bottom shows 'Double digits' and a message indicating the program finished with exit code 0.

```
main.rb
1 def average(arr)
2   # Check if the array is empty to avoid division by zero
3   return 0 if arr.empty?
4
5   sum = 0
6   num_elements = arr.length
7
8   # Use a while loop to calculate the sum of all elements in the array
9   i = 0
10  while i < num_elements
11    sum += arr[i]
12    i += 1
13  end
14
15  # Calculate the average by dividing the sum by the number of elements
16  average = sum.to_f / num_elements
17
18  return average
19 end
20
21 numbers = [10, 20, 30, 40, 50]
22 result = average(numbers)
23
24 if result >= 10
25   puts "Double digits"
26 else
27   puts "Single digits"
28 end
```

input

Double digits

...Program finished with exit code 0  
Press ENTER to exit console.