Advanced Scripting   
Program Flow Control Structures

Last Updated: 1/3/2024 5:24 PM Version 1  
Document Prepared for: CYBER360 Student

# Name Click here to enter name ID Click here to enter id

# Instructions

Answer all questions directly in this document. You will save and upload this completed document as your homework submission.

# Overview

For this exercise, you will write a few short scripts to practice using PowerShell control structures.

# Setup

## Requirements

* Visual Studio Code, PowerShell

# Task 1—The **if** Statement

The most used conditional execution statement starts with the **if** keyword. You will use it all the time. *(Note: if you have programmed in any C-based languages, this will be easy to pick up since the syntax is quite similar. The only real difference is that curly braces {} are always required in PowerShell.*

## Steps

1. Simple **if**. The if statement executes the block of code after the condition if the expression evaluates to **True**, but skips it if the expression evaluates to **False**.
   1. Open VSCode and enter the following script:

$yours=Read-Host "What is your favorite color?"

$mine='blue'

if ($yours -eq $mine) {

Write-Host 'Our favorite colors are the same!'

}

* 1. Run the script [F5] several times, entering green, blue, and Blue for your favorite color. What did you learn about case sensitivity?Click or tap here to enter text.

1. **else**. In the previous step the script produces output only when the colors match. It would be better if we told the user when they did not match as well. The else block executes only if the if block does not.
   1. Add the following lines to the bottom of your script:

else {

"We don't have the same favorite color."

}

* 1. Run the script again a couple of times, entering blue and green to view the results

1. **elseIf**. Sometimes you want to check several conditions. The elseif keyword can be used to chain if statements together. If the first condition does not evaluate **True**, the elseif condition is evaluated. If that evaluates **True** then its statement block is executed. You can have as many elseif statements as you need. The final else block only gets executed if none of the if or elseif conditions are **True**.
   1. Enter the following code:

$answer=Get-Random -Minimum 1 -Maximum 10$

$guess=Read-Host "What is your guess?"

if ($answer -gt $guess) {

"Higher"

} elseif ($answer -lt $guess){

"Lower"

} else {

"Correct"

}

* 1. Run the code a few times to see the results. To “cheat” and spy on the script’s random **$answer** while it’s running:
     1. Save your code to a script file (such as **C:\TEMP\ex2.4.ps1**).
     2. Before you run the code, set a breakpoint [F9] at the **$guess=** statement.
     3. Then when you run [F5] your code, it will *pause* before asking for your guess.
     4. You can look at the value of the local variable **$answer** in the top-left pane of VSCode’s debugger tool (or you can enter **$answer** in the terminal pane).
     5. With that knowledge, you’ll know what guess you should type to make the script execute whichever conditional script block you want to test.
     6. Continue [F5] to resume execution, and the script will finally prompt you to enter your guess.

# Task 2—The Switch Statement

The switch statement is used when you want to test multiple conditions. Each pattern in the switch statement is tested, and if it matches, that pattern’s statement block is executed. Multiple blocks may match. Use the **break** statement if you don’t want the rest of the tests performed. The **default** block is only executed if there are no matches.

## Steps

1. You will now make a mini-menu system using **switch** to act on the user’s input.
   1. Enter the following code in a new script:

$command=Read-Host @'

What would you like to do?

1. Start

2. Stop

3. Continue

4. Quit

Enter Choice

'@

switch ($command){

'1' {'Starting'}

'2' {'Stopping'}

'3' {'Continuing'}

'4' {'Quitting'}

default {'Invalid Command'}

}

* 1. Run the program a few times entering 1, 2, 3, 4 or 5to see the output.

1. Now you will allow the user to type the menu word as well as the number.
   1. Modify your switch statement to look like this:

switch ($command){

'1' {'Starting'}

'start' {'Starting'}

'2' {'Stopping'}

'stop' {'Stopping'}

'3' {'Continuing'}

'continue' {'Continuing'}

'4' {'Quitting'}

'quit' {'Quitting'}

default {'Invalid Command'}

}

* 1. Run your script. Enter numbers (1, 2, 3, 4, 5) or words (start, stop, continue, quit, delay) to get a feel for how it works.

1. Typing *continue* is too long and annoys the users. Let’s fix it so you only need to type the first letter, using pattern matches.
   1. Modify your **switch** statement to look like this

switch -Wildcard ($command){

'1' {'Starting'}

's\*' {'Starting'}

'2' {'Stopping'}

's\*' {'Stopping'}

'3' {'Continuing'}

'c\*' {'Continuing'}

'4' {'Quitting'}

'q\*' {'Quitting'}

default {'Invalid Command'}

}

* 1. Run the code. Try 1, s, start, q, etc. Record your observations. Click or tap here to enter text.

1. Since the switch statement executes *all* blocks that match the pattern, when you entered **‘s’** or **‘st’** it executed both start and stop outputs. To fix that you could just fix the matching patterns so the user has to enter enough of the word to uniquely identify the options.
   1. Fix it by modifying your code as follows:

switch -Wildcard ($command){

'1' {'Starting'}

'sta\*' {'Starting'}

'2' {'Stopping'}

'sto\*' {'Stopping'}

'3' {'Continuing'}

'c\*' {'Continuing'}

'4' {'Quitting'}

'q\*' {'Quitting'}

default {'Invalid Command'}

}

* 1. Try it again and observe the results.

1. You can replace a pattern with an expression enclosed in curly braces **{}**. The expressions should each evaluate as either **True** or **False**. In each expression, the automatic variable **$\_** holds the value of the expression or command that followed the **switch** keyword.
   1. In the previous example we used two lines of code to provide two different pattern matches. You can create an expression to do the same thing in one line. Example: the start options could be rewritten as:  
      **{$\_ -eq '1' -or $\_ -like 'sta\*'} {'Starting'}**
   2. Rewrite the switch statement using similar conditions for all options. Enter your script here:
   3. Click or tap here to enter text.
2. Since the switch statement executes all matches if you only want the first match executed you must use the break statement.
   1. Try this in a new script:

switch (1){

1 {'this matches'}

2 {'this doesn''t'}

1 {'so does this'}

}

* 1. Run the script. Notice both lines that have 1 are executed.
  2. Use the **break** statement to tell the switch operator to quit processing other matches. Add a break statement:

switch (1){

1 {

'this matches'

break

}

2 {'this doesn''t'}

1 {'so does this'}

}

* 1. Run the script again. Record the output: Click or tap here to enter text.

# Task 3—Foreach\_Object loop cmdlet, .foreach() object method

## Steps

1. At a PowerShell prompt, Enter:  
   **1..4 | Foreach-Object { "Progress: $\_ of 4" }**
   1. Do it again, but this time use the shorter alias **%** instead of **Foreach-Object**:  
      **1..4 | % {"Progress: $\_ of 4"}**
   2. Modify the string in that pipeline’s script block, so that it produces output that looks like this:  
      **Progress: 25%  
      Progress: 50%  
      Progress: 75%  
      Progress: 100%**
      1. Enter your modified string here:  
         **1..4 | % {**Click or tap here to enter text.**}**
2. At a PowerShell prompt, Enter:  
   **(1..4).foreach({ "Progress: $\_ of 4" })**
   1. Modify the string in that method’s script block argument, so that its output looks like this:  
      **Remaining: 3 of 4  
      Remaining: 2 of 4  
      Remaining: 1 of 4  
      Remaining: 0 of 4**
      1. Enter your modified string here:  
         **(1..4).foreach({**Click or tap here to enter text.**})**

# Task 4—Loop control statements: while, do, foreach, for

## Steps

1. Back in VS Code, copy the following code skeleton outline into a new script:  
   **$i = \_  
   while ( \_ ) {  
    \_  
   }**
   1. Everywhere you see a single underscore \_, replace with suitable code to make the script produce this output:  
      **Number 5  
      Number 6  
      Number 7  
      Number 8  
      Number 9  
      Number 10**
   2. Enter your working code snippets here:  
      **$i =** Click or tap here to enter text. **while (** Click or tap here to enter text. **) {** Click or tap here to enter text. **}**
2. Add the following code skeleton outline:  
   **$i = \_  
   do {  
    \_  
   } while ( \_ )**
   1. Everywhere you see a single underscore \_, replace it with suitable code to produce this output:  
      **Number 100  
      Number 110  
      Number 120  
      Number 130  
      Number 140  
      Number 150**
   2. Enter your working code snippets here:  
      **$i =** Click or tap here to enter text. **do {** Click or tap here to enter text. **} while (** Click or tap here to enter text. **)**
3. Add the following code skeleton outline:  
   **$i = \_  
   do {  
    \_  
   } until ( \_ )**
   1. Everywhere you see a single underscore \_, replace it with suitable code to produce this output:  
      **Number 128  
      Number 256  
      Number 384  
      Number 512  
      Number 640  
      Number 768**
   2. Enter your working code snippets here:  
      **$i =** Click or tap here to enter text. **do {** Click or tap here to enter text. **} until (** Click or tap here to enter text. **)**
4. Add the following code skeleton outline:  
   **for ( \_ ; \_ ; \_ ) {  
    \_  
   }**
   1. Everywhere you see a single underscore \_, replace it with suitable code to produce this output:  
      **Number -9  
      Number -12  
      Number -15  
      Number -18  
      Number -21  
      Number -24**
   2. Enter your working code snippets here:  
      **for (** Click or tap here to enter text. **;** Click or tap here to enter text. **;** Click or tap here to enter text. **) {** Click or tap here to enter text. **}**
5. Add the following code skeleton outline:  
   **$listing = Get-ChildItem  
   foreach ( \_ in $listing ) {  
    \_  
   }**
   1. Everywhere you see a single underscore \_, replace it with suitable code that outputs the name and the size (in kilobytes) of the files (but not the subdirectories) in the current directory.
   2. Enter your working code snippets here:  
      **$listing = Get-ChildItem  
      foreach (** Click or tap here to enter text. **in $listing ) {** Click or tap here to enter text. **}**
6. Use any loop statement of your choice to write code that outputs the numbers 3 through 12, but skips every third number, like this:  
   **Number 3  
   Number 4  
   Number 6  
   Number 7  
   Number 9  
   Number 10  
   Number 12**
   1. Enter your working looping code here:  
       Click or tap here to enter text.

# Deliverable

Upload this document with completed answers to I-Learn Canvas.