

# Basic Of Shell Scripting

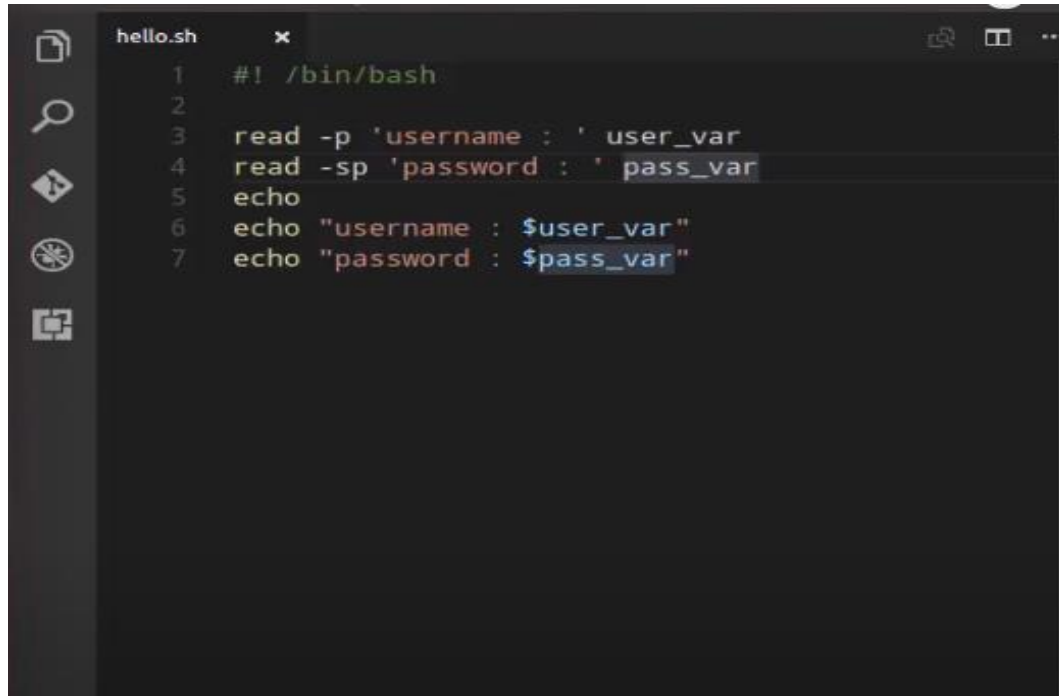
## II

# read Command

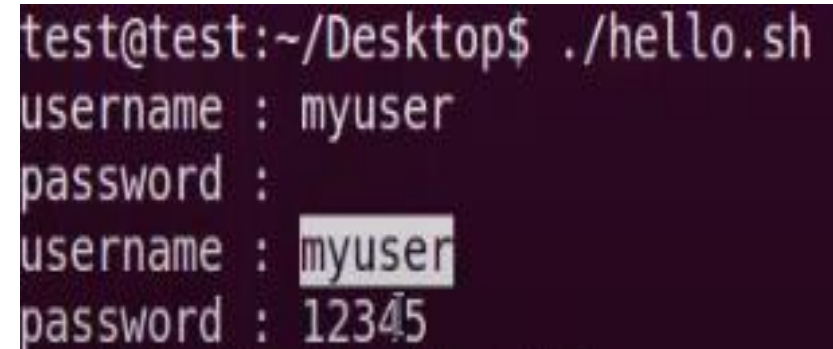
- Syntax for taking multiple user input using read keyword.

```
#!/bin/bash
echo "Enter three course name"
read name1 name2 name3
echo "names are: $name1, $name2, $name3"
```

# Example

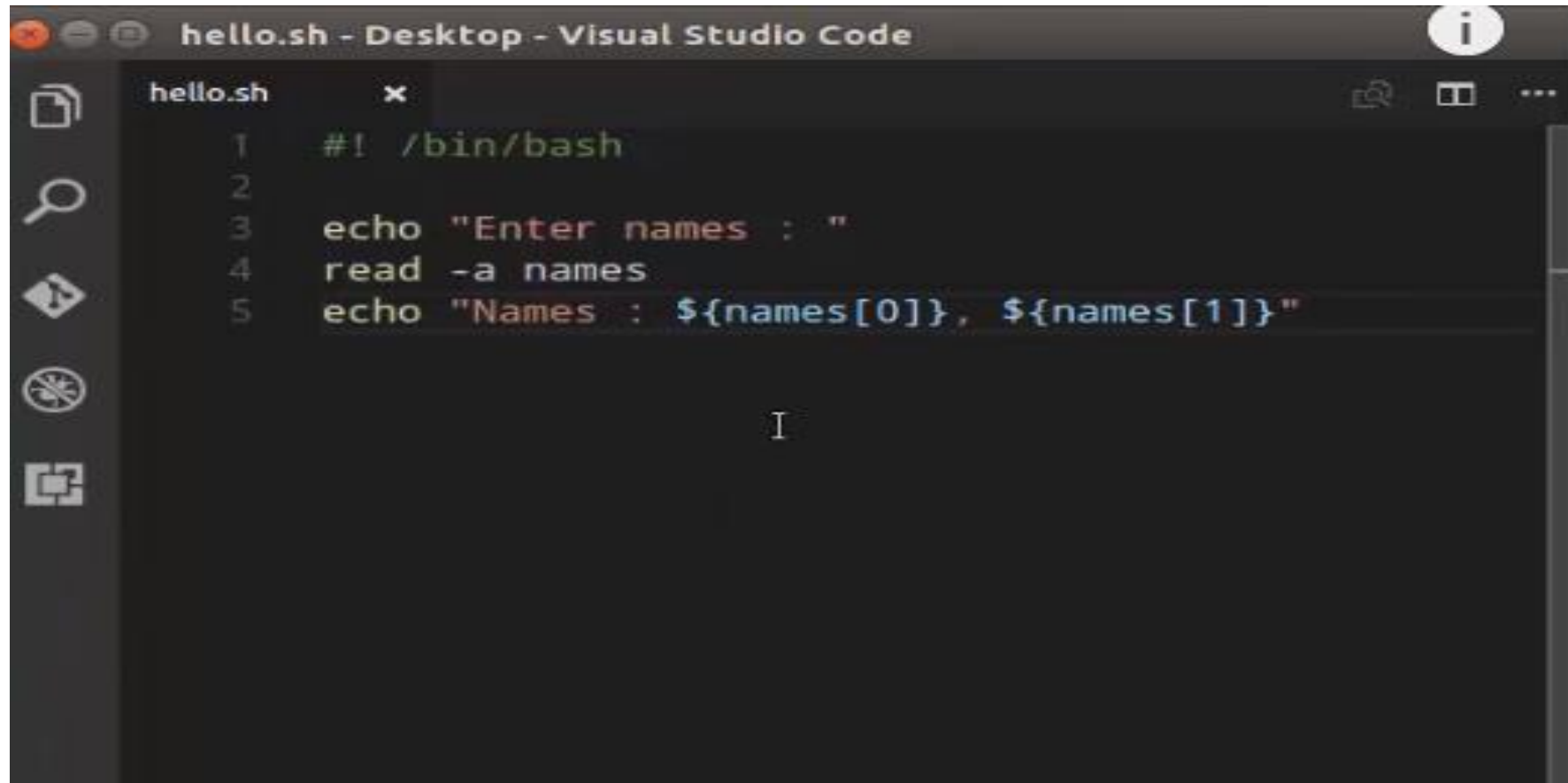
A screenshot of a code editor window titled 'hello.sh'. The editor contains a shell script with seven lines of code. Line 1 is the shebang '#!/bin/bash'. Line 2 is empty. Line 3 is 'read -p 'username : ' user\_var'. Line 4 is 'read -sp 'password : ' pass\_var'. Line 5 is 'echo'. Line 6 is 'echo "username : \$user\_var"'. Line 7 is 'echo "password : \$pass\_var"'. The code is syntax-highlighted with green for the shebang, red for prompts, blue for variables, and black for strings and echo. A left sidebar contains icons for file operations, search, and other editor functions.

```
hello.sh
1  #!/bin/bash
2
3  read -p 'username : ' user_var
4  read -sp 'password : ' pass_var
5  echo
6  echo "username : $user_var"
7  echo "password : $pass_var"
```

A screenshot of a terminal window showing the execution of the script. The prompt is 'test@test:~/Desktop\$'. The command './hello.sh' has been entered. The output shows 'username : myuser' and 'password : ' followed by a new line. Then 'username : myuser' and 'password : 12345' are displayed. The input 'myuser' and '12345' are highlighted with a light blue selection box.

```
test@test:~/Desktop$ ./hello.sh
username : myuser
password :
username : myuser
password : 12345
```

# Example

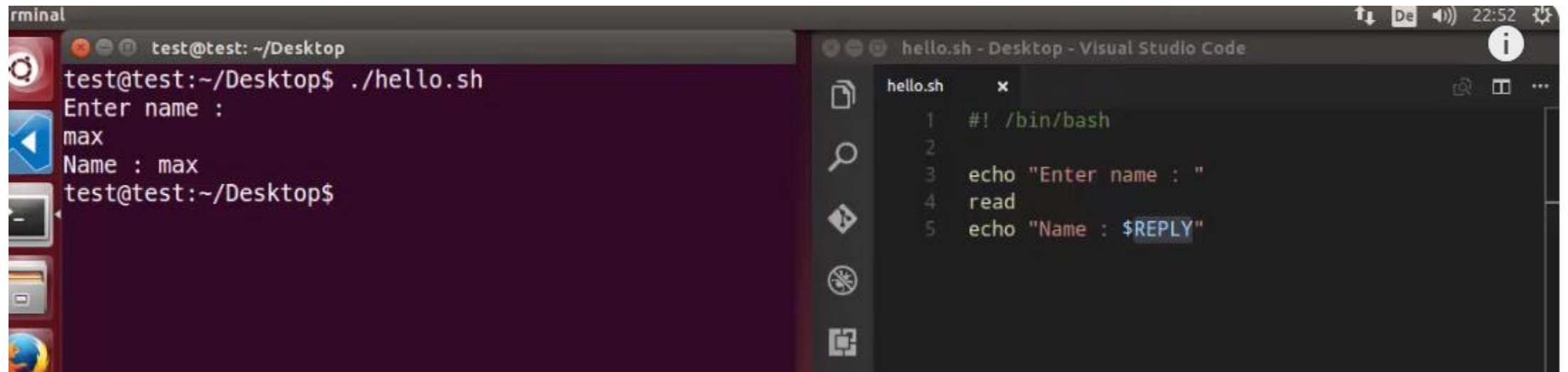


The image shows a screenshot of a Visual Studio Code editor window. The title bar at the top reads "hello.sh - Desktop - Visual Studio Code". The editor is displaying a shell script file named "hello.sh". The script contains five lines of code, which are numbered 1 through 5 on the left margin. The code is as follows:

```
1  #! /bin/bash
2
3  echo "Enter names : "
4  read -a names
5  echo "Names : ${names[0]}, ${names[1]}"
```

The script is a bash script that prompts the user to enter names and then displays the entered names. The first line is the shebang line, indicating the script is written for the bash shell. The second line is empty. The third line uses the `echo` command to print the prompt "Enter names : ". The fourth line uses the `read -a names` command to read input from the user into an array named `names`. The fifth line uses the `echo` command to print the names stored in the array, separated by a comma and a space, using the syntax `${names[0]}` and `${names[1]}` to access the first and second elements of the array. The cursor is positioned at the end of the fifth line.

# Example



The image shows a screenshot of a computer screen with two windows. The left window is a terminal titled 'rminal' with the prompt 'test@test: ~/Desktop'. It shows the execution of a script: 'test@test:~/Desktop\$ ./hello.sh', followed by the prompt 'Enter name :', the input 'max', and the output 'Name : max'. The right window is a Visual Studio Code editor titled 'hello.sh - Desktop - Visual Studio Code'. It shows the contents of the 'hello.sh' script:

```
hello.sh  x
1  #! /bin/bash
2
3  echo "Enter name : "
4  read
5  echo "Name : $REPLY"
```

The status bar at the top right of the VS Code window shows 'De', a speaker icon, and the time '22:52'.

# Adding Basic Options

- Let's take a look at some of the most basic options we can use:
- *-a array*: stores the results of the word split operation in an array rather than separate variables
- *-s*: does not echo the input line to the standard output stream
- *-p prompt*: print the prompt text before requesting the input from the standard input stream without a *<newline>* character
- *-t timeout*: attempt to read the input for a given period of seconds
- *-N*: read exactly N characters from the input unless a timeout occurs or *EOF* is reached

# Conditional Statement

## If-else Statement #

- Bash if conditionals can have different forms. The most basic if statement takes the following form:

if TEST-COMMAND

then

STATEMENTS

Else

STATEMENTS

fi

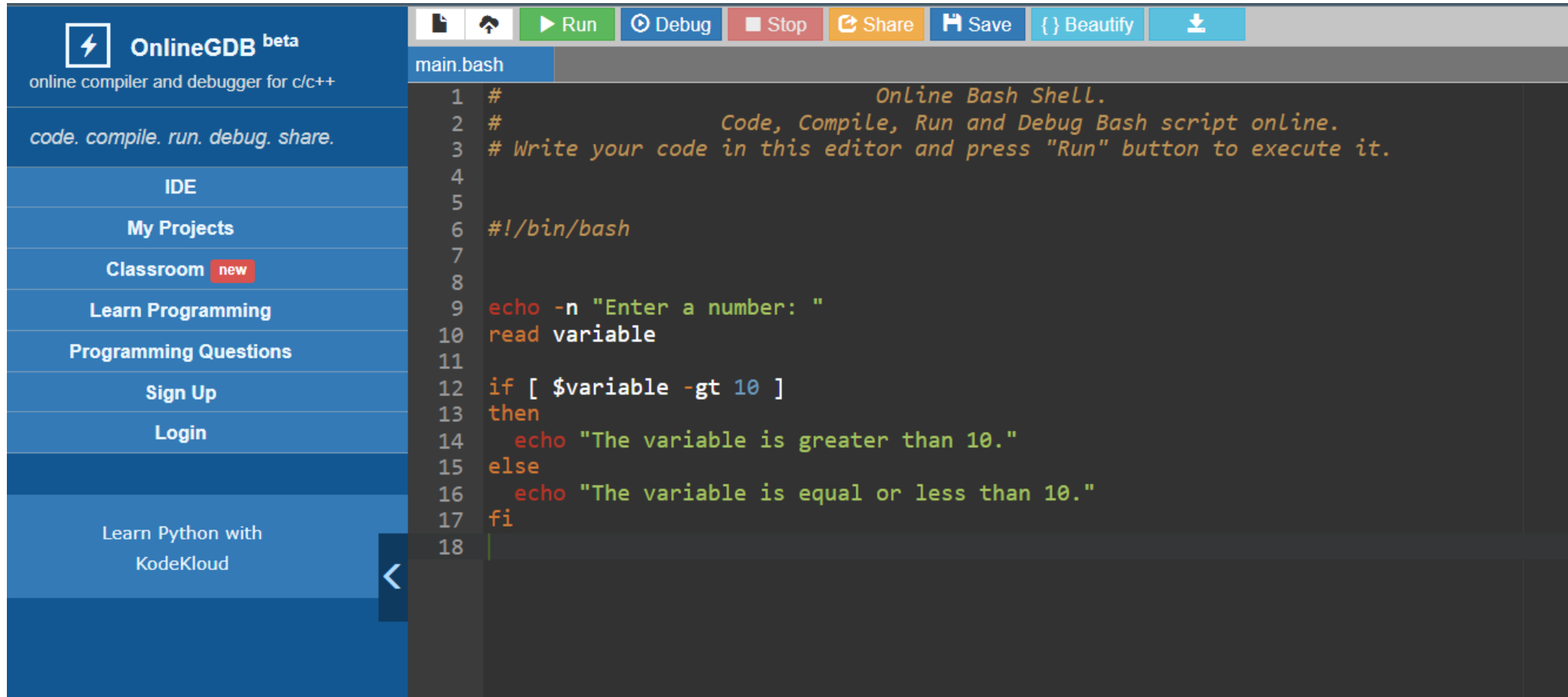
- The if statement starts with the if keyword followed by the conditional expression and the then keyword. The statement ends with the fi keyword.

# Syntax

```
untitled — Atom
untitled • Telemetry Consent Welcome

1 integer comparison
2
3 -eq - is equal to - if [ "$a" -eq "$b" ]
4 -ne - is not equal to - if [ "$a" -ne "$b" ]
5 -gt - is greater than - if [ "$a" -gt "$b" ]
6 -ge - is greater than or equal to - if [ "$a" -ge "$b" ]
7 -lt - is less than - if [ "$a" -lt "$b" ]
8 -le - is less than or equal to - if [ "$a" -le "$b" ]
9 < - is less than - (( "$a" < "$b" ))
10 <= - is less than or equal to - (( "$a" <= "$b" ))
11 > - is greater than - (( "$a" > "$b" ))
12 >= - is greater than or equal to - (( "$a" >= "$b" ))
13
14 string comparison
15 = - is equal to - if [ "$a" = "$b" ]
16 == - is equal to - if [ "$a" == "$b" ]
17 != - is not equal to - if [ "$a" != "$b" ]
18 < - is less than, in ASCII alphabetical order - if [[ "$a" < "$b" ]
19 > - is greater than, in ASCII alphabetical order - if [[ "$a" > "$b" ]
20 -z - string is null, that is, has zero length
```

# Example Code



The screenshot displays the OnlineGDB web interface. On the left is a blue sidebar with navigation links: OnlineGDB beta, code. compile. run. debug. share., IDE, My Projects, Classroom (with a 'new' badge), Learn Programming, Programming Questions, Sign Up, Login, and Learn Python with KodeKloud. The top of the editor features a toolbar with icons for file operations and buttons for Run, Debug, Stop, Share, Save, Beautify, and Download. The main editor area shows a file named 'main.bash' containing a Bash script. The script includes comments about the Online Bash Shell and a program that prompts the user for a number, then checks if it is greater than 10 or less than/equal to 10, printing the result accordingly.

```
1  #                               Online Bash Shell.
2  #                               Code, Compile, Run and Debug Bash script online.
3  # Write your code in this editor and press "Run" button to execute it.
4
5
6  #!/bin/bash
7
8
9  echo -n "Enter a number: "
10 read variable
11
12 if [ $variable -gt 10 ]
13 then
14     echo "The variable is greater than 10."
15 else
16     echo "The variable is equal or less than 10."
17 fi
18
```

# if..elif..else Statement

```
if TEST-COMMAND1
then
    STATEMENTS1
elif TEST-COMMAND2
then
    STATEMENTS2
else
    STATEMENTS3
fi
```

If the TEST-COMMAND1 evaluates to True, the STATEMENTS1 will be executed. If the TEST-COMMAND2 evaluates to True, the STATEMENTS2 will be executed. If none of the test commands evaluate to True, the STATEMENTS2 is executed.

# Example Code



The screenshot displays the OnlineGDB beta web interface. On the left is a dark blue sidebar with navigation links: 'IDE', 'My Projects', 'Classroom' (with a red 'new' badge), 'Learn Programming', 'Programming Questions', 'Sign Up', 'Login', and 'Learn Python with KodeKloud'. The main area features a toolbar with icons for file operations and buttons for 'Run' (green), 'Debug' (blue), 'Stop' (red), 'Share' (orange), 'Save' (blue), 'Beautify' (light blue), and a download icon. Below the toolbar, a tab labeled 'main.bash' is active, showing a Bash script with line numbers 1 through 20. The script includes comments about the Online Bash Shell and a logic to check if a variable is greater than, equal to, or less than 10.

```
1 # Online Bash Shell.
2 # Code, Compile, Run and Debug Bash script online.
3 # Write your code in this editor and press "Run" button to execute it.
4
5
6 #!/bin/bash
7
8
9 echo -n "Enter a number: "
10 read Variable
11
12 if [ $Variable -gt 10 ]
13 then
14     echo "The variable is greater than 10."
15 elif [ $Variable -eq 10 ]
16 then
17     echo "The variable is equal to 10."
18 else
19     echo "The variable is less than 10."
20 fi
```

# Arithmetic Operation

```
main.bash
3  # Write your code in this editor and press "Run" button to execute it.
4  #!/bin/bash
5  x=10
6  y=20
7  echo "x=10, y=5"
8  echo "Addition of x and y"
9  echo $(( $x + $y ))
10 echo "Subtraction of x and y"
11 echo $(( $x - $y ))
12 echo "Multiplication of x and y"
13 echo $(( $x * $y ))
14 echo "Division of x by y"
15 echo $(( $x / $y ))
16 echo "Exponentiation of x,y"
17 echo $(( $x ** $y ))
18 echo "Modular Division of x,y"
19 echo $(( $x % $y ))
20 echo "Incrementing x by 10, then x= "
21 (( x += 10 ))
22 echo $x
23 echo "Decrementing x by 15, then x= "
24 (( x -= 15 ))
25 echo $x
26 echo "Multiply of x by 2, then x="
27 (( x *= 2 ))
28 echo $x
29 echo "Dividing x by 5, x= "
30 (( x /= 5 ))
31 echo $x
32 echo "Remainder of Dividing x by 5, x="
33 (( x %= 5 ))
34 echo $x
35
```

# Arithmetic Operation

```
input
x=10, y=5
Addition of x and y
30
Subtraction of x and y
-10
Multiplication of x and y
200
Division of x by y
0
Exponentiation of x,y
7766279631452241920
Modular Division of x,y
10
Incrementing x by 10, then x=
20
Decrementing x by 15, then x=
5
Multiply of x by 2, then x=
10
Dividing x by 5, x=
2
Remainder of Dividing x by 5, x=
2

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

# Floating Point Operation

```
#!/bin/bash

# Define floating-point numbers
num1=3.14
num2=2.5

# Use bc with -l option for floating-point arithmetic and math functions
result=$(echo "scale=2; $num1 + $num2" | bc -l)

# Print the result
echo "The sum is: $result"
```

# Explanation

- `scale=2;` sets the number of decimal places to 2 in the result. You can adjust the scale value based on your precision requirements.
- `echo "scale=2; $num1 + $num2" | bc -l` performs the floating-point arithmetic with the specified precision.
- Again, this is just a simple example. You can modify the expression within `echo` to perform other mathematical operations or use additional functions provided by the `bc` math library when using the `-l` option.
- This output represents the result of the addition operation  $3.14 + 2.5$  with a scale of 2 decimal places, as specified in the `scale=2` setting. The `bc -l` option allows for floating-point arithmetic and makes the math library available for use in the expression.