**LINUX SHELL**

The file /etc/shells gives an overview of known shells on a Linux system. (/etc/shells dosyası, bir Linux sistemindeki bilinen kabuklara genel bir bakış sunar.)

cat /etc/shells

/bin/bash

/bin/sh

/bin/tcsh

/bin/csh

Your default shell is set in the /etc/passwd file.( Varsayılan kabuğunuz /etc/passwd dosyasında ayarlanır.)

The easiest way is to use which command: $ which bash

Create a folder and create a file in it. Note all the scripts would have the **.sh extension. Note:** Bash ignores everything written on the line after the hash mark `(#)`. The only exception to this rule is the first line of the script that starts with the `#!`(shebang) characters.

mkdir shell-scripting && cd shell-scripting

Create a `script` file named `basic.sh`

#!/bin/bash

echo “Hello World”

chmod +x basic.sh

./basic.sh or bash basic.sh

**“Hello World”**

A shell script may be made executable by using the chmod command to turn on the execute bit. When Bash finds such a file while searching the PATH for a command, it spawns a sub-shell to execute it.( Yürütme bitini açmak için chmod komutu kullanılarak bir kabuk betiği yürütülebilir hale getirilebilir . Bash, PATH'de bir komut ararken böyle bir dosya bulduğunda, onu yürütmek için bir alt kabuk oluşturur.) `./` means we're calling something in the current working directory. ./filename.sh or bash filename.sh

**Here is our first bash shell script example; traditional hello world script:**

#!/bin/bash

# declare STRING variable

STRING="Hello World"

#print variable on a screen

echo $STRING

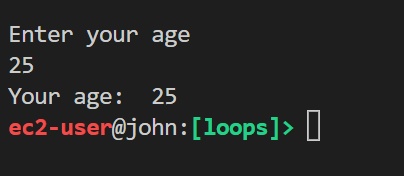
$ chmod +x hello\_world.sh

$ ./hello\_world.sh

**Hello World!**

**Let's request user's age then print his age:**

#! /bin/bash

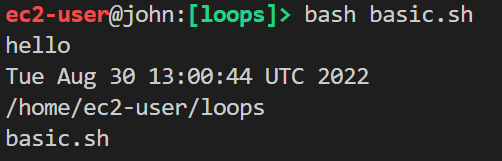
clear

echo "Enter your age"

read st1

echo "Your age: " $st1

**$ ./your\_age.sh**

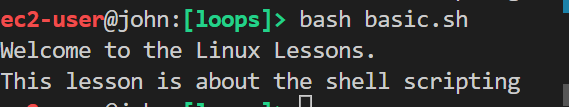
#!/bin/bash

echo "hello"

date

pwd

ls

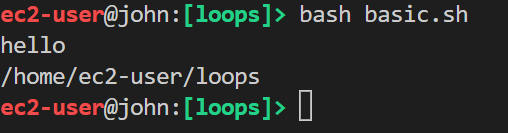
#!/bin/bash

cat << EOF

Welcome to the Linux Lessons.

This lesson is about the shell scripting

EOF

#!/bin/bash

echo "hello"

# date

pwd # This is an inline comment

# ls

**Shell Variables**

The name of a variable can contain only letters (a to z or A to Z), numbers ( 0 to 9) or the underscore character (\_) and beginning with a letter or underscore character.

- The following examples are valid variable names.

```bash

KEY=value

\_VAR=5

**clarus\_way=test**

**```**

**> Note** that there is no space on either side of the equals ( = ) sign.

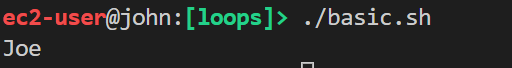
- The following examples are invalid.

```bash

3\_KEY=value

-VAR=5

clarus-way=test

KEY\_1?=value1

#!/bin/bash

NAME=Joe

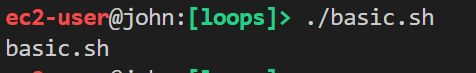
echo $NAME

**### Command Substitution**

Command substitution empowers us to take the output of a command or program (which would usually be written on the screen) and save it as the value of a variable. To do this we put it inside brackets, followed by a $ symbol.

#!/bin/bash

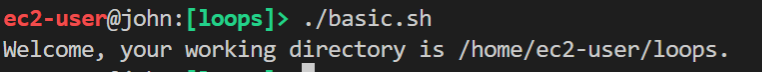
content=$(ls) or we can use `(backtick) content=`ls`

****echo $content

**let's see that in a script**.

working\_directory=$(pwd)

echo "Welcome, your working directory is $working\_directory."



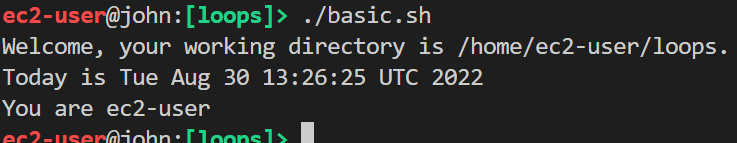
We can also get same result without using variables.

#!/bin/bash

echo "Welcome, your working directory is $(pwd)."

echo "Today is `date`"

echo "You are `whoami`"



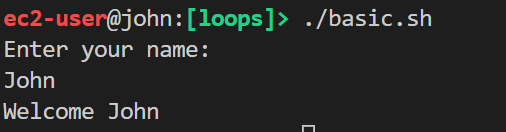
The Bash `read` command is a powerful built-in utility used take user input. (Bash `read` komutu, kullanıcı girdisi almak için kullanılan güçlü bir yerleşik yardımcı programdır.)

#!/bin/bash

echo "Enter your name: "

read NAME

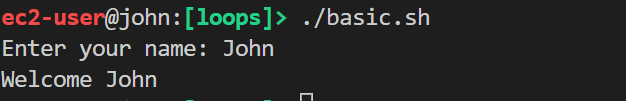
echo "Welcome $NAME"



When writing interactive bash scripts, we can use the read command to get the user input. To specify a prompt string, use the -p option. The prompt is printed before the read is executed and doesn’t include a newline.( Etkileşimli bash betikleri yazarken, kullanıcı girdisini almak için read komutunu kullanabiliriz. Bir bilgi istemi dizesi belirtmek için -p seçeneğini kullanın. Bilgi istemi, okuma yürütülmeden önce yazdırılır ve yeni bir satır içermez.)

#!/bin/bash

read -p "Enter your name: " NAME

echo "Welcome $NAME"

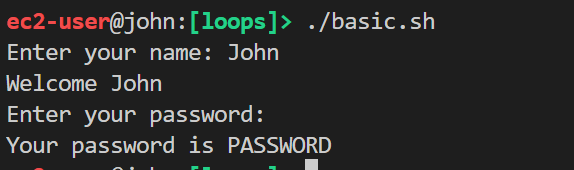
When entering sensitive information we do not want to display input coming. For this we can use `read -s`

read -p "Enter your name: " NAME

echo "Welcome $NAME"

read -s -p "Enter your password: " PASSWORD

echo -e "\nYour password is $PASSWORD"



**### Command Line Arguments**

Command-line arguments are given after the name of the program in command-line shell of Operating Systems. The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

#!/bin/bash

echo "File Name is $0"

echo "First Parameter is $1"

echo "Second Parameter is $2"

echo "Third Parameter is $3"

echo "All the Parameters are $@"

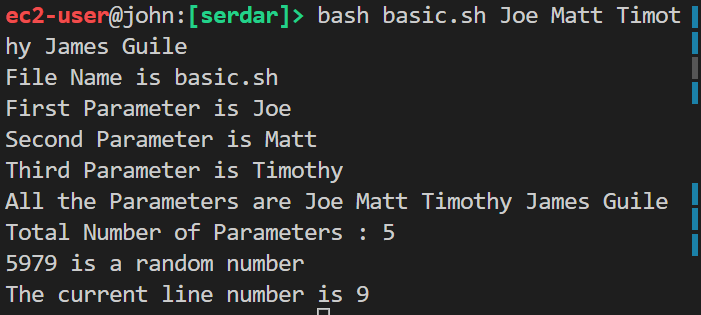
echo "Total Number of Parameters : $#"

echo "$RANDOM is a random number"

echo "The current line number is $LINENO"

chmod +x basic.sh (Aynı dosyada çalışıyorsanız gerek yok.)

./basic.sh Joe Matt Timothy James Guile



**### Arrays**

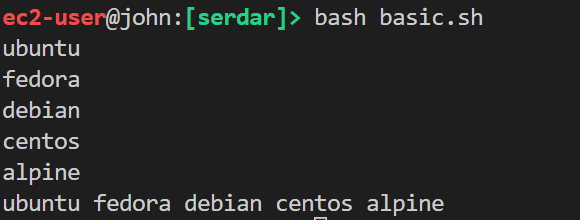
- In our programs, we usually need to group several values to render as a single value. In shell, arrays can hold multiple values at the same time.

**#### Defining arrays**

- Following is the simplest method of creating an array variable.

#!/bin/bash

DISTROS[0]="ubuntu"

DISTROS[1]="fedora"

DISTROS[2]="debian"

DISTROS[3]="centos"

DISTROS[4]="alpine"

echo ${DISTROS[0]}

echo ${DISTROS[1]}

echo ${DISTROS[2]}

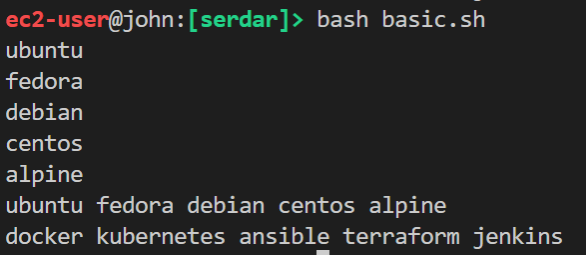
echo ${DISTROS[3]}

echo ${DISTROS[4]}

echo ${DISTROS[@]}

#!/bin/bash

SERDAR[0]="ubuntu"

SERDAR[1]="fedora"

SERDAR[2]="debian"

SERDAR[3]="centos"

SERDAR[4]="alpine"

echo ${SERDAR[0]}

echo ${SERDAR[1]}

echo ${SERDAR[2]}

echo ${SERDAR[3]}

echo ${SERDAR[4]}

echo ${SERDAR[@]}

devops\_tools=("docker" "kubernetes" "ansible" "terraform" "jenkins")

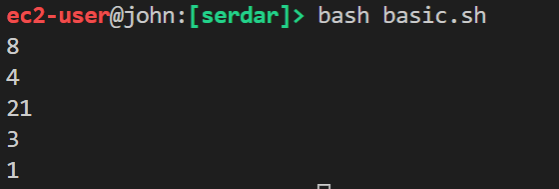
echo "docker kubernetes ansible terraform jenkins"

**## - Simple Arithmetic**

 There are many ways to evaluate arithmetic expression in Bash scripting

**### expr**

- `expr` command print  the value of expression to standard output. Let's see this.

#!/bin/bash

expr 3 + 5

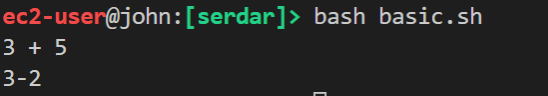
expr 6 - 2

expr 7 \\* 3

expr 9 / 3

expr 7 % 2

Using `expr` command, we must have spaces between the items of the expression and must not put quotes around the expression. If we do that, the expression will not be evaluated but printed instead. See the difference.

#!/bin/bash

expr "3 + 5"

expr 3-2

#!/bin/bash

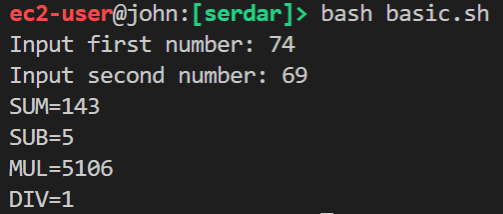
read -p "Input first number: " first\_number

read -p "Input second number: " second\_number

echo "SUM="`expr $first\_number + $second\_number`

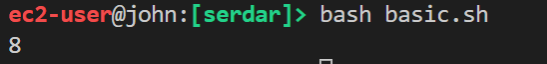
echo "SUB="`expr $first\_number - $second\_number`

echo "MUL="`expr $first\_number \\* $second\_number`

echo "DIV="`expr $first\_number / $second\_number`

**## let**

- `let` is a builtin function of Bash that helps us to do simple arithmetic. It is similar to `expr` except instead of printing the answer it saves the result to a variable. Unlike expr we need to enclose the expression in quotes.

#!/bin/bash

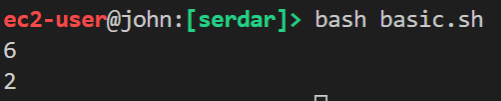
let "sum = 3 + 5"

echo $sum

We can also increase or decrease the variable by 1 with `let` function. Let's see this.

#!/bin/bash

x=5

let x++

echo $x

y=3

let y--

echo $y

Create a file and name it `let-calculator.sh`.

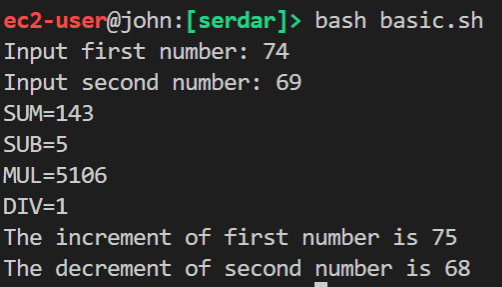
#!/bin/bash

read -p "Input first number: " first\_number

read -p "Input second number: " second\_number

let "sum = $first\_number + $second\_number"

let "sub = $first\_number - $second\_number"

let "mul = $first\_number \* $second\_number"

let "div = $first\_number / $second\_number"

echo "SUM=$sum"

echo "SUB=$sub"

echo "MUL=$mul"

echo "DIV=$div"

let first\_number++

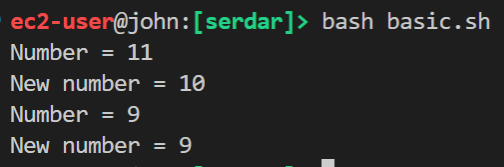
let second\_number--

echo "The increment of first number is $first\_number"

echo "The decrement of second number is $second\_number"

#!/bin/bash

number=10

let new\_number=number++   # This firstly assigns the number then increases.

echo "Number = $number"

echo "New number = $new\_number"

number=10

let new\_number=--number   # This firstly decreases the number then assigns.

echo "Number = $number"

echo "New number = $new\_number"

**#### Difference between `num++` and `++num`, or `num--` and `--num`**

number=10

let new\_number=number++   # This firstly assigns the number then increases.

echo "Number = $number"

echo "New number = $new\_number"



number=10

let new\_number=--number   # This firstly decreases the number then assigns.

echo "Number = $number"

echo "New number = $new\_number"



#!/bin/bash

read -p "Input first number: " first\_number

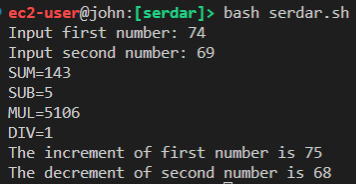
read -p "Input second number: " second\_number

sum=$(($first\_number + $second\_number))

sub=$(($first\_number - $second\_number))

mul=$(($first\_number \* $second\_number))

div=$(($first\_number / $second\_number))



echo "SUM=$sum"

echo "SUB=$sub"

echo "MUL=$mul"

echo "DIV=$div"

(( first\_number++ ))

(( second\_number-- ))

echo "The increment of first number is $first\_number"

echo "The decrement of second number is $second\_number"

**### Double Parentheses**

- We can also evaluate arithmetic expression with double parentheses. We have learned that we could take the output of a command and save it as the value of a variable. We can use this method to do basic arithmetic.

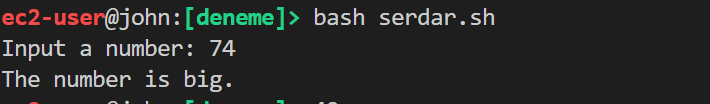
#!/bin/bash

sum=$((3 + 5))

echo $sum

#!/bin/bash

read -p "Input a number: " number

if [[ $number -gt 50 ]]

then

  echo "The number is big."

fi

We can use `Relational Operators`, `String Operators` or `File Test Operators` inside the square brackets ( [ ] ) in the if statement above.

**### Relational Operators**

- Bourne Shell supports the relational operators below that are specific to numeric values. These operators do not work for string values.

| Operator | Description |

| -------- | ----------- |

| -eq   | equal                  |

| -ne   | not equal              |

| -gt   | greater than           |

| -lt   | less than              |

| -ge   | greater than or equal  |

| -le   | less than or equal     |

**### String Operators**

- The string operators below are supported by Bourne Shell.

| Operator | Description |

| -------- | ----------- |

| =    | equal            |

| !=   | not equal        |

| -z   | Empty string     |

| -n   | Not empty string |

#!/bin/bash

if [[ "a" = "a" ]]

then

  echo "They are same"

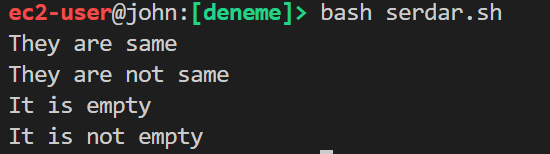
fi

if [[ "a" != "b" ]]

then

  echo "They are not same"

fi

if [[ -z "" ]]

then

  echo "It is empty"

fi

if [[ -n "text" ]]

then

  echo "It is not empty"

fi

Notice that there are spaces between the opening bracket `[` and the parameters "text" = "text", and then between the parameters and the closing bracket `]`. That is precisely because the brackets here act as a command, and you are separating the command from its parameters.

**### File Test Operators**

- There are a few operators that can be used to test various properties associated with a Linux file.

| Operator | Description |

| -------- | ----------- |

| -d file   | directory  |

| -e file   | exists     |

| -f file   | ordinary file     |

| -r file   | readable          |

| -s file   | size is > 0 bytes |

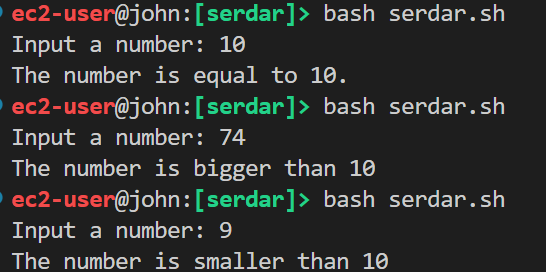
| -w file   | writable          |

| -x FILE   | executable        |

**## - If Elif Else Statements**

- The elif statement is used when it requires to specify several conditions in our program.

#!/bin/bash

read -p "Input a number: " number

if [[ $number -eq 10 ]]

then

  echo "The number is equal to 10."

elif [[ $number -gt 10 ]]

then

  echo "The number is bigger than 10"

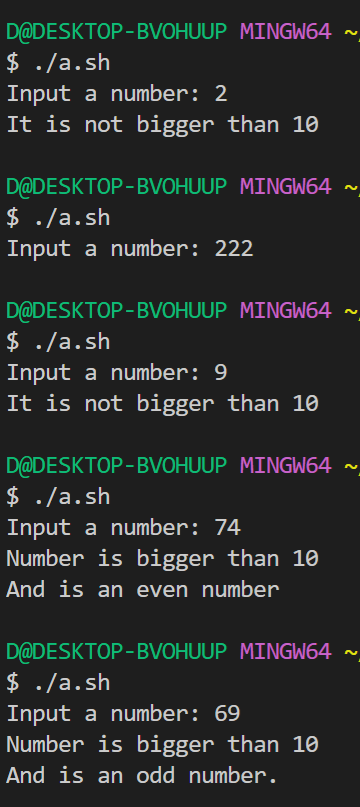
else

  echo "The number is smaller than 10"

fi

**## Part 4 - Nested If Statements**

- If statements can be nested. Let's see the nested structure on the followig example.

#!/bin/bash

read -p "Input a number: " number

if [[ $number -gt 10 ]]

then

  echo "Number is bigger than 10"

  if (( $number % 2 == 1 ))

  then

    echo "And is an odd number."

  else

    echo "And is an even number"

  fi

else

  echo "It is not bigger than 10"

fi

**## - Boolean Operations**

- The Boolean operators below are supported by the Bourne Shell.

| Operator | Description |

| -------- | ----------- |

| !        | negation    |

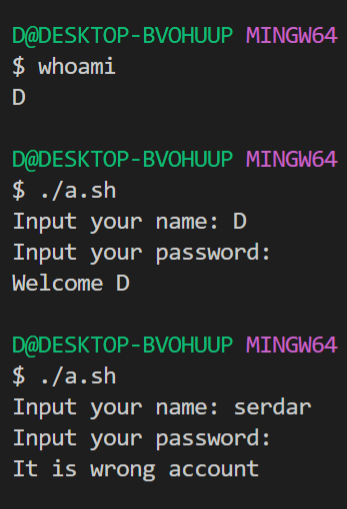
| &&       | and         |

| ||       | or          |

- `!`  inverts a true condition into false and vice versa.

- `&&` is logical AND. If both the operands are true, then the condition becomes true otherwise false.

- `||`  is logical OR. If one of the operands is true, then the condition becomes true.

#!/bin/bash

read -p "Input your name: " name

read -sp "Input your password: " password

if [[ $name = $(whoami) ]] && [[ $password = Aa1234 ]]

then

  echo -e "\nWelcome $(whoami)"

else

  echo -e "\nIt is wrong account"

fi

#!/bin/bash

read -p "Input first number: " first\_number

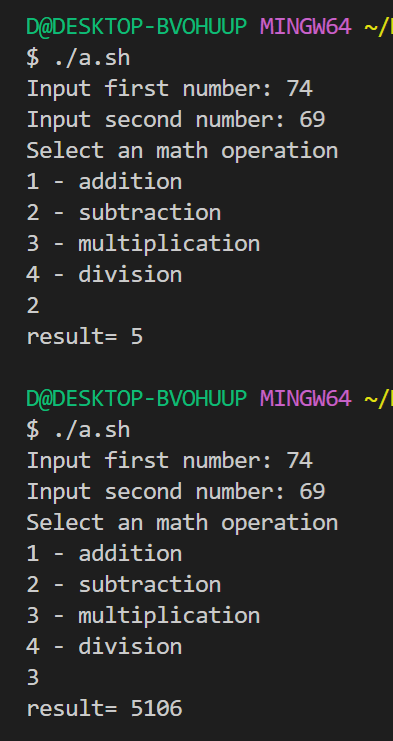
read -p "Input second number: " second\_number

read -p "Select an math operation

1 - addition

2 - subtraction

3 - multiplication

4 - division

" operation

case $operation in

  "1")

     echo "result= $(( $first\_number + $second\_number))"

  ;;

  "2")

     echo "result= $(( $first\_number - $second\_number))"

  ;;

  "3")

     echo "result= $(( $first\_number \* $second\_number))"

     ;;

  "4")

     echo "result= $(( $first\_number / $second\_number))"

  ;;

  \*)

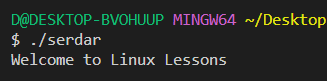
     echo "Wrong choice..."

  ;;

esac

It is pretty easy to declare and call a function.

#!/bin/bash



Welcome () {

    echo "Welcome to Linux Lessons"

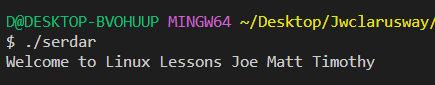
}

Welcome

#!/bin/bash

Welcome () {

    echo "Welcome to Linux Lessons $1 $2 $3"

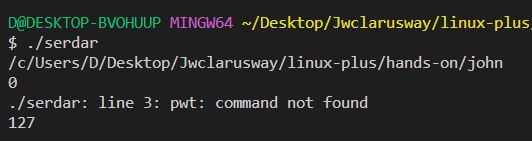
}

Welcome Joe Matt Timothy

```

**## - Returning Values from Functions**

- Functions in other programming languages return a value when called. But, Bash functions don’t return a value when called. But we can define a return status similar to exit status of a command.

- When any shell command terminates, it returns an exit code, which indicates `0` for success and non-zero decimal number in the `1 - 255` range for failure. The special variable `$?` returns the exit status of the last executed command. Let's see this.

pwd

echo $?  #0

pwt  # It is wrong command

echo $?  #127

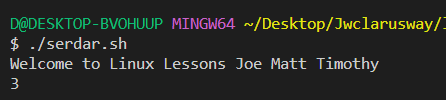
```

When a bash function completes, its return value is the status of the last statement executed in the function. We can speciy return status by using the `return` keyword. We can think the `return` keyword as exit status of function.

#!/bin/bash

Welcome () {

    echo "Welcome to Linux Lessons $1 $2 $3"

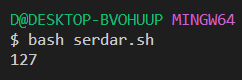
    return 3

    }

Welcome Joe Matt Timothyecho $?

echo $?

#!/bin/bash

Welcome () {

    if [[ $1 -eq 10 ]]

    then

    return 0

    else

    return 127

    fi

    }

Welcome 12

echo $?

**## - Nested Functions**

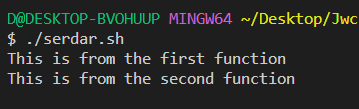
- One of the useful features of functions is that they can call themselves and other functions.

#!/bin/bash

function\_one () {

   echo "This is from the first function"

   function\_two

}

function\_two () {

   echo "This is from the second function"

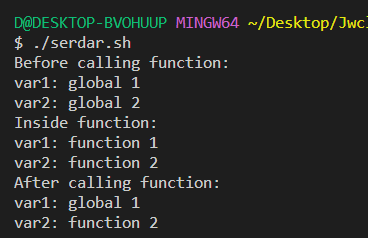
}

function\_one

**## - Variables Scope**

- Global variables are variables that can be accessed from anywhere in the script regardless of the scope. In Bash, by default all variables are defined as global, even if declared inside the function.

Local variables can be declared within the function body with the local keyword and can be used only inside that function.



!/bin/bash

var1='global 1'

var2='global 2'

var\_scope () {

  local var1='function 1'

  var2='function 2'

  echo -e "Inside function:\nvar1: $var1\nvar2: $var2"

}

echo -e "Before calling function:\nvar1: $var1\nvar2: $var2"

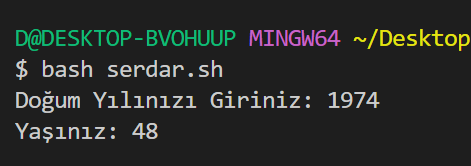
var\_scope

echo -e "After calling function:\nvar1: $var1\nvar2: $var2"

#!/bin/bash #thanks for this script to F4512-John

function print\_age(){

    read -p "Doğum Yılınızı Giriniz: " birth\_year

    let yas=$1-$birth\_year

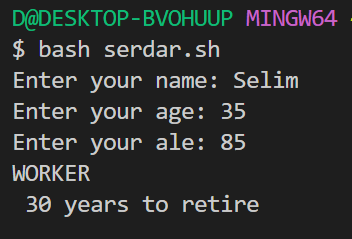
    echo "Yaşınız: $yas"

}

print\_age 2022

#thanks for this script to F3696- Fatih



#!/bin/bash

read -p "Enter your name: " NAME

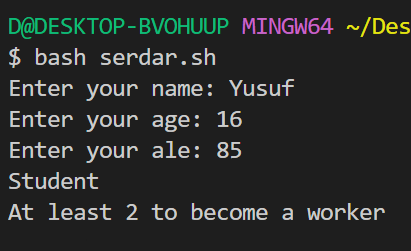
read -p "Enter your age: " AGE

read -p "Enter your ale: " ALE

if [[ $AGE -lt 18 ]]

then

echo "Student"

 echo "At least $((18-AGE)) to become a worker"

elif [[ $AGE -gt 18 && $AGE -lt 65 ]]

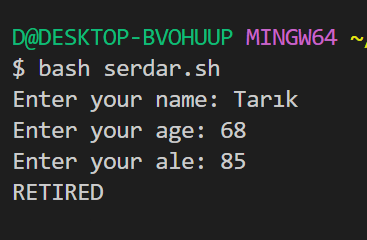
then

echo "WORKER"

echo " $((65-AGE)) years to retire"

else # age greater than 65

if [[ $AGE -lt $ALE ]]

 then

echo "RETIRED"

else

echo "BEEP SOUND"

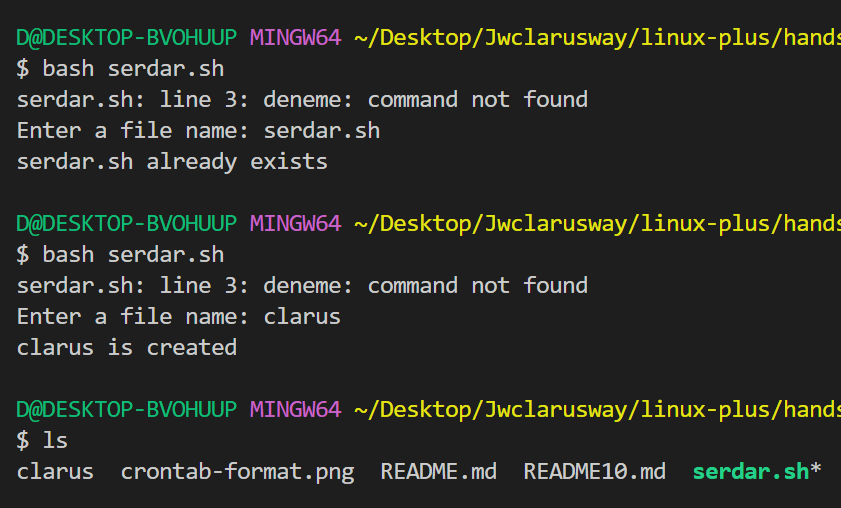
echo "!!! ALREADY DIED !!!"

echo "wait 1 sec."

fi

fi

#!/bin/bash



deneme 2

read -p "Enter a file name: " FILE

if [[ -e $FILE ]]

then

  echo "$FILE already exists"

else

  touch $FILE

  echo "$FILE is created"

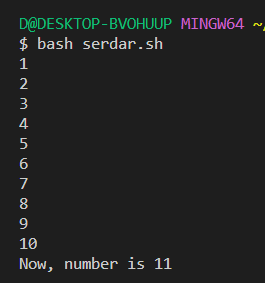
fi

**Shell Scripting/Loops**

## Part 1 - While loops

- When writing programs in shell, in some cases it is not enough to execute our block of code only once. The loops are used to repeat (iterate) the execution of a block of code.

- while loops have a boolean logic, similar to if statements. As long as the result of the condition returns True, the code block under while loop runs. When th - while loops have a boolean logic, similar to if statements. As long as the result of the condition returns True, the code block under while loop runs. When the condition returns to False, the loop execution is terminated and the program control moves further to the next operation.

#!/bin/bash

number=1

while [[ $number -le 10  ]]

do

  echo $number

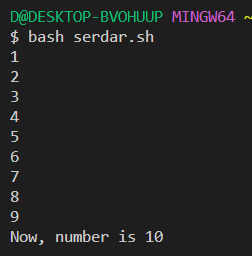
  ((number++))

done

echo "Now, number is $number"

**## Part 2 - Until loops**

- The until loop is identical to the while loop, except that it will execute the commands within it until the test becomes true.

#!/bin/bash

number=1

until [[ $number -ge 10  ]]

do

  echo $number

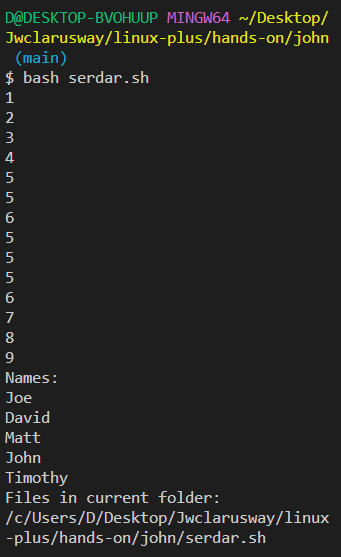
  ((number++))

done

echo "Now, number is $number"

**## Part 3 - For loops**

- Sometimes we want to iterate a block of code for each of the items in a given list. For this we use `for loop`.



#!/bin/bash

echo "Numbers:"

for number in 0 1 2 3 4 5 6 7 8 9

do

   echo $number

done

echo "Names:"

for name in Joe David Matt John Timothy

do

   echo $name

done

echo "Files in current folder:"

for file in `pwd`/\*

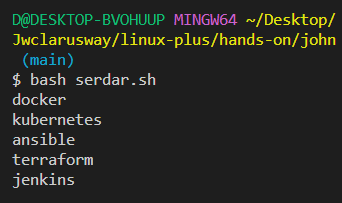
do

   echo $file

done

**### Using arrays with the for loop**

#!/bin/bash

devops\_tools=("docker" "kubernetes" "ansible" "terraform" "jenkins")

for tool in ${devops\_tools[@]}

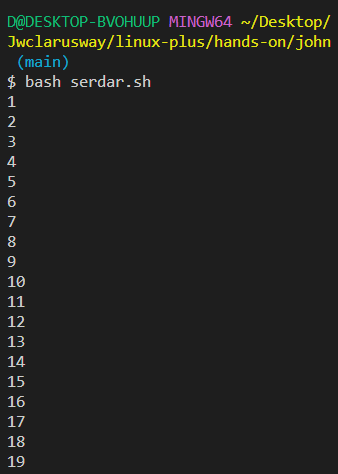
do

   echo $tool

done

**## Part 4 - Continue and Break Statements**

As we see above the infinite loop run forever. The break statement is used to terminate the execution of the entire loop.( Yukarıda gördüğümüz gibi sonsuz döngü sonsuza kadar çalışır. Break ifadesi, tüm döngünün yürütülmesini sonlandırmak için kullanılır.)

#!/bin/bash

number=1

until [[ $number -lt 1  ]]

do

  echo $number

  ((number++))

  if [[ $number -eq 20 ]]

  then

    break

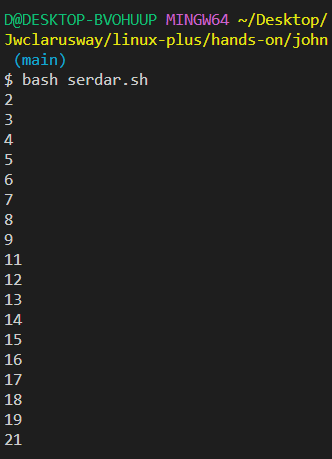
  fi

done

**### Continue Statement**

- The Continue statement is similar to the Break command, except it causes the current iteration of the loop to exit, instead of the whole loop.( - Continue ifadesi, tüm döngü yerine döngünün geçerli yinelemesinin çıkmasına neden olması dışında Break komutuna benzer.)

#!/bin/bash

number=1

until [[ $number -lt 1  ]]

do

  ((number++))

  tens=$(($number % 10))

  if [[ $tens -eq 0 ]]

  then

    continue

  fi

  echo $number

  if [[ $number -gt 20 ]]

  then

    break

  fi

done

**## Part 5 - Select loops**

- The Select Loop generates a numbered menu from which users can select options. It's helpful when you need to ask the user to select one or more items from a list of options.

#!/bin/bash

read -p "Input first number: " first\_number

read -p "Input second number: " second\_number

PS3="Select the operation: "

select operation in addition subtraction multiplication division exit

do

  case $operation in

    addition)

      echo "result= $(( $first\_number + $second\_number))"

    ;;

    subtraction)

       echo "result= $(( $first\_number - $second\_number))"

    ;;

    multiplication)

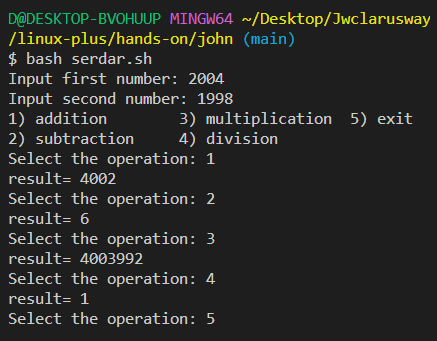
       echo "result= $(( $first\_number \* $second\_number))"

       ;;

    division)

       echo "result= $(( $first\_number / $second\_number))"

    ;;

    exit)

       break

    ;;

    \*)

       echo "Wrong choice..."

    ;;

  esac

done

Exercises

#for döngüsü örnekleri:

#! /bin/bash

#Listedeki tüm elemanları ekrana yazdır:

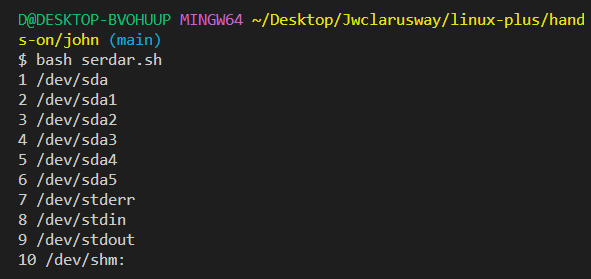
for oyuncu in Pele Maradona Platini Zidane Ronaldinho "Roberto Carlos" Ronaldo Messi

do

  echo "Döneminin En İyisi: $oyuncu"

done



#Wildcard kullanım örneği:

#! /bin/bash

counter=1

for dosya in $(ls /dev/s\*)

do

  echo "$counter $dosya"

  let counter=counter+1

done

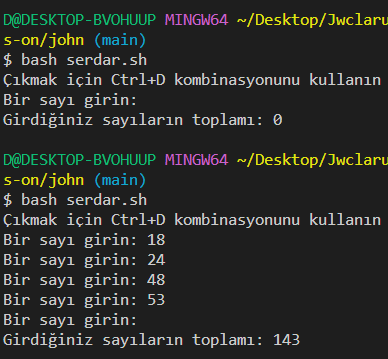
#! /bin/bash

#Kullanıcıdan alınan sayıların toplamı:

#! /bin/bash

toplam=0

echo "Çıkmak için Ctrl+D kombinasyonunu kullanın"

printf "Bir sayı girin: "

while read sayi

do

  let toplam=toplam+sayi

  printf "Bir sayı girin: "

done

echo ""

echo "Girdiğiniz sayıların toplamı: $toplam"

#!/bin/bash

TL=35000

DOLLAR=3500

EURO=5000

read -p "Enter Amount of Money You Want to Withraw: " Money

read -p "Select an Currency :

1 - TL

2 - DOLLAR

3 - EURO

" operation

case $operation in

"1")

symbol="₺"

echo "Account= $(( $TL ))" $symbol

if [[ $Money -gt $TL ]]

then

echo "Your Account Isn't Enough"

echo "Have A Nice Day"

date

elif [[ $Money -le $TL ]]

then

echo "Your New Account = "`expr $TL - $Money` "$symbol"

echo "Have A Nice Day"

date

fi

;;

"2")

symbol="$"

echo "Account= $(( $DOLLAR ))" $symbol

if [[ $Money -gt $DOLLAR ]]

then

echo "Your Account Isn't Enough"

echo "Have A Nice Day"

date

elif [[ $Money -le $DOLLAR ]]

then

echo "Your New Account = "`expr $DOLLAR - $Money` "$symbol"

echo "Have A Nice Day"

date

fi

;;

"3")

symbol="€"

echo "Account= $(( $EURO ))" $symbol

if [[ $Money -gt $EURO ]]

then

echo "Your Account Isn't Enough"

echo "Have A Nice Day"

date

elif [[ $Money -le $EURO ]]

then

echo "Your New Account = "`expr $EURO - $Money` "$symbol"

echo "Have A Nice Day"

date

fi

;;

esac