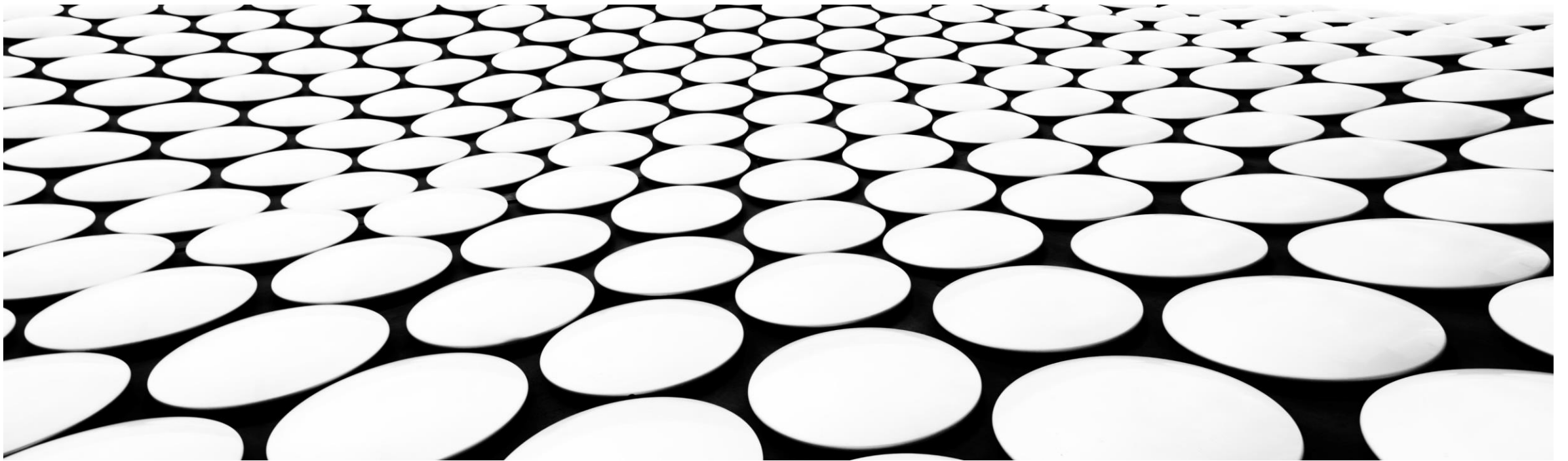


---

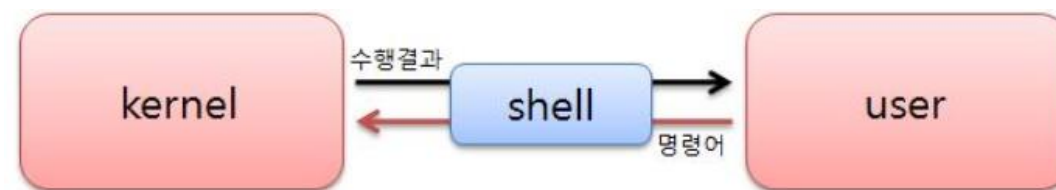
# LINUX의 셸 스크립트 및 MAKEFILE



# LINUX - SHELL SCRIPT

## Shell 이란?

- 셸(Shell)은 커널(Kernel)과 사용자간의 다리 역할을 하는 것으로 사용자로부터 명령을 받아 그것을 해석하고 프로그램을 실행하는 역할을 한다.
- 셸은 사용자가 시스템에 로그인하게 되면 각 사용자에게 설정된 셸이 부여 되면서 다양한 명령어를 수행할 수 있게 된다. 달리 말하면 사용자에게 셸을 부여하지 않게 되면 시스템에 로그인하더라도 명령을 수행할 수 없게 되므로 로그인을 막는 효과와 동일하다고 볼 수 있다.



# LINUX - SHELL SCRIPT

## 셸의 역사

- 리눅스의 모태가 되는 유닉스 최초의 셸은 켄 톰프슨(Ken Thompson)이 멀틱스(Multics) 셸을 따라 모형화한 셸을 이용하였고, 그 후 스티븐 본(Steven Bourne)이 유닉스 버전 7의 기본 셸이 되는 본 셸(Bourne Shell, sh)을 개발하였다. 본 셸은 강력한 셸이었지만 유용한 기능이 많지 않았다.
- 그 후, 버클리 대학의 빌 조이(Bill Joy)가 개발한 C 셸(C shell, csh)이 등장하였으며 현재에는 bash, ksh, tcsh, zsh과 같은 다양한 셸이 개발 되었다. 리눅스에는 sh를 기본으로 ksh와 csh 계열의 장점을 결합한 bash shell(Bourne Again shell)을 표준으로 하고 있다.

# LINUX - SHELL SCRIPT

## 셸의 종류

### ■ bourne 셸 계열의 셸

- 1. sh (bourne shell)
- 가장 기본적인 셸로 유닉스의 초기부터 사용되어 온 셸이다. 스크립트를 지원한다.
- 2. ksh (korn shell)
- 본 셸을 확장한 셸이다. 본 셸의 명령어를 모두 인식하며, 명령어 히스토리(history) 기능과 앨리어스(alias), 작업 제어 등의 기능이 추가되었다. 일반적으로 유닉스에서 가장 많이 사용되는 셸이다. 명령행 편집기능을 제공한다.
- 3. bash (Bourne Again Shell)
- 리눅스에서 가장 많이 사용하는 셸이다. C 셸과 콘 셸의 장점을 결합하여 작성되었으며, Bourne 셸 문법의 명령어 셋을 제공하여 Bourne Shell과 호환되는 셸로 GNU 프로젝트에 의해 만들어지고 배포된다. 명령행 편집기능을 제공한다.

### ■ C 셸 계열의 셸

- 1. csh (C Shell)
- 명령행 편집기능을 제공하지 않는다. C 언어 위주의 셸로 처음 작성되었을 때에는 본 셸이 가지고 있지 못한 기능들(작업제어, 명령어 히스토리 등)을 가지고 있었기 때문에 많이 사용되었다.
- 2. tcsh (TC Shell)
- csh의 기능을 강화한 셸이다. 확장 C Shell. 명령행 편집 기능을 제공한다.
- CentOS, SUSE Linux, Asianux 등의 레드햇 계열의 리눅스 배포판에서 기본으로 사용되지 않는 셸
- Zsh 셸
- 로그인셸 및 셸스크립트 명령어 프로세서로서 이용 가능한 유닉스 셸이다. 표준 셸들 중에서 zsh는 ksh와 가장 유사하지만 많은 개선들을 포함한다. Zsh는 명령행 편집, 내장 스펠링 수정, history 등의 기능을 가진다.
- Ash 셸
- 추가적인 기능들이 없이 본 셸에 가장 부합하는 셸이다. 본 셸은 상업적인 유닉스 시스템들에서 사용 가능하므로, ash는 셸 스크립트가 본 셸에 잘 부합하는지 시험할 때 유용하다. 또한 이것은 다른 sh- 호환 셸에 비해 적은 메모리와 공간을 요구한다.

# LINUX - SHELL SCRIPT

## bash shell 이란

- bash 셸은 1989년 브라이언 폭스(Brian Fox)가 GNU 프로젝트를 위해 개발하였으며 본 셸(Bourne Shell)을 기반으로 만들어졌다.
- GNU 운영체제, 리눅스, 맥OS X 등 다양한 운영체제에서 사용 중이며 현재 리눅스의 표준 셸이다. bash의 명령어 문법은 sh와 호환되고 ksh와 csh의 유용한 기능을 참고하여 명령 히스토리, 명령어 완성 기능, 히스토리 치환, 명령행 편집 등을 지원하고 있다.

# LINUX - SHELL SCRIPT

## bash shell 예약 변수

- 쉘 스크립트에서 사용자가 정해서 만들 수 없는 이미 정의된 변수가 존재한다. 그것을 예약 변수라고 한다. 몇가지 예약 변수를 보도록 합시다.
- Linux의 shell은 xterm에서 구동된다.

변수	설명
HOME	사용자 홈 디렉토리를 의미한다.
PATH	실행 파일의 경로이다. 여러분이 chmod, mkdir 등의 명령어들은 /bin이나 /usr/bin, /sbin에 위치하는데, 이 경로들을 PATH 지정하면 여러분들은 굳이 /bin/chmod를 입력하지 않고, chmod 입력만 해주면 된다.
LANG	프로그램 실행 시 지원되는 언어를 말한다.
UID	사용자의 UID이다.
SHELL	사용자가 로그인시 실행되는 쉘을 말한다.
USER	사용자의 계정 이름을 말한다.
FUNCNAME	현재 실행되고 있는 함수 이름을 말한다.
TERM	로그인 터미널을 말한다.

# LINUX - SHELL SCRIPT

## bash shell 프롬프트 구성

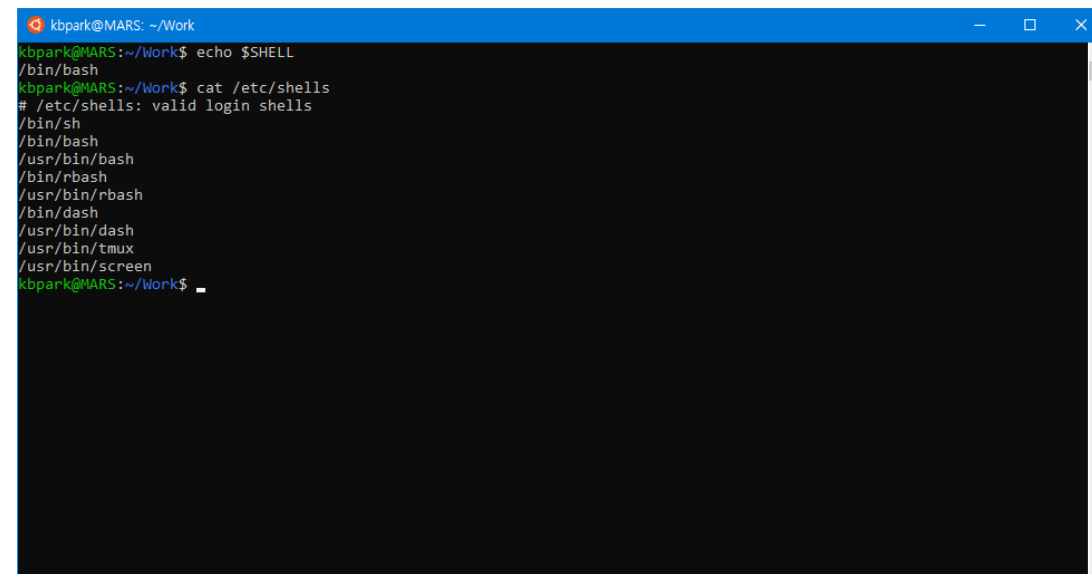
- 리눅스환경에서는 사용자마다 다른 셸을 지정할 수 있다. bash shell에 접속하게 된다면 일반적으로 보게 될 화면을 다음과 같다.

```
kbpark@MARS: ~/Work
systemd-timesync:x:102:104:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:103:106:/:nonexistent:/usr/sbin/nologin
syslog:x:104:110:/home/syslog:/usr/sbin/nologin
_apt:x:105:65534:/:nonexistent:/usr/sbin/nologin
tss:x:106:111:TPM software stack,,,:/var/lib/tpm:/bin/false
uidd:x:107:112:/run/uidd:/usr/sbin/nologin
tcpdump:x:108:113:/:nonexistent:/usr/sbin/nologin
sshd:x:109:65534:/run/sshd:/usr/sbin/nologin
landscape:x:110:115:/var/lib/landscape:/usr/sbin/nologin
pollinate:x:111:1:/var/cache/pollinate:/bin/false
kbpark:x:1000:1000:,,,:/home/kbpark:/bin/bash
kbpark@MARS:~$ ls
Download Work litecoin-api-exam.tar.gz
kbpark@MARS:~$ cd Work/
kbpark@MARS:~/Work$ ls
bitcoin facebook_react litecoin litecoin_0.13 litecoin_0.15 makecoin node_exam shell_exam solidity_exam
kbpark@MARS:~/Work$ ls -al
total 44
drwxr-xr-x 11 kbpark kbpark 4096 May 18 11:15 .
drwxr-xr-x 16 kbpark kbpark 4096 May 25 21:09 ..
drwxr-xr-x 14 kbpark kbpark 4096 Apr 1 10:35 bitcoin
drwxr-xr-x 9 kbpark kbpark 4096 May 18 11:16 facebook_react
drwxr-xr-x 14 kbpark kbpark 4096 Mar 18 11:05 litecoin
drwxr-xr-x 11 kbpark kbpark 4096 Mar 18 11:07 litecoin_0.13
drwxr-xr-x 12 kbpark kbpark 4096 Mar 18 11:08 litecoin_0.15
drwxr-xr-x 8 kbpark kbpark 4096 May 7 10:42 makecoin
drwxr-xr-x 4 kbpark kbpark 4096 May 14 11:34 node_exam
drwxr-xr-x 2 kbpark kbpark 4096 May 7 10:53 shell_exam
drwxr-xr-x 6 kbpark kbpark 4096 Apr 9 17:57 solidity_exam
kbpark@MARS:~/Work$
```

# LINUX - SHELL SCRIPT

## 셸의 확인

- 시스템에 로그인한 후에 사용 중인 셸을 확인하려면 로그인 셸 관련 환경변수 SHELL을 이용해서 가능하다. 즉 아래와 같은 명령문으로 확인할 수 있다.
  - `$ echo $SHELL`
- 시스템에 사용 가능한 셸의 리스트를 확인 하려면 다음과 같은 명령문으로 확인할 수 있다.
  - `$ cat /etc/shells`



```
kbpark@MARS: ~/Work
kbpark@MARS:~/Work$ echo $SHELL
/bin/bash
kbpark@MARS:~/Work$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
/usr/bin/tmux
/usr/bin/screen
kbpark@MARS:~/Work$
```



# LINUX - SHELL SCRIPT

## 셸의 변경

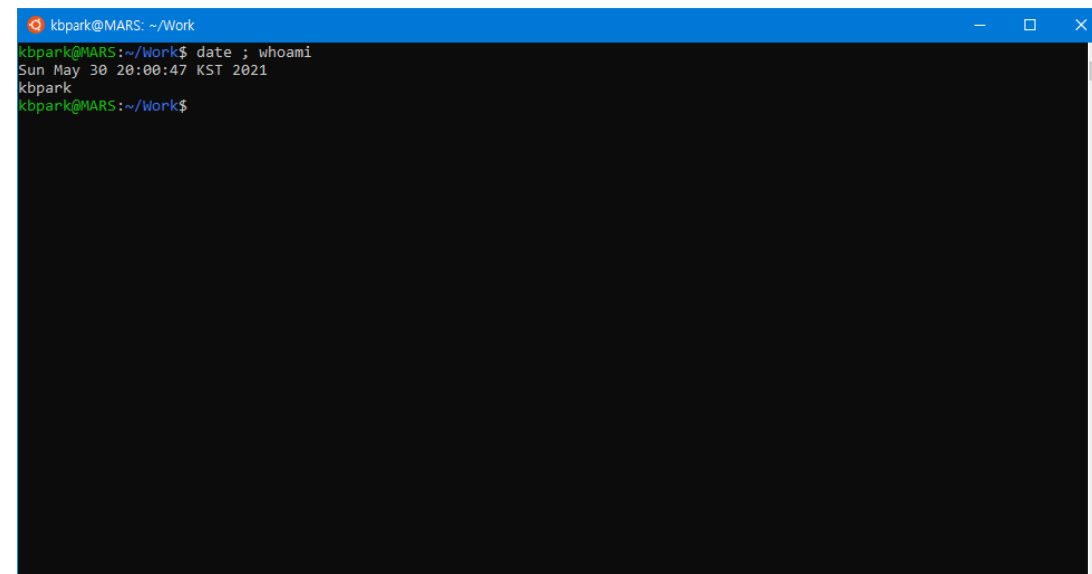
- 셸 리스트 중 현재와 다른 셸을 사용하려면 'chsh'라는 명령어를 이용하여 셸의 절대경로를 입력하면 된다.
- 변경한 셸은 다음 로그인부터 적용된다.
- 다음은 sh 셸에서 bash 셸로 바꾼 예이다.
- 최근 mac은 zsh 셸을 사용 하고 있다.

```
# echo $SHELL
/bin/sh
# chsh
Changing the login shell for root
Enter the new value, or press ENTER for the default
    Login Shell [/bin/sh]: /bin/bash
#
test1@ServerTest2:~$ su -
Password:
root@ServerTest2:~# echo $SHELL
/bin/bash
root@ServerTest2:~#
```

# LINUX - SHELL SCRIPT

## 여러 명령 사용

- 세미콜론(;)
  - 하나의 라인에 주어진 명령어들을 성공,실패와 관련 없이 전부 실행한다.
  - \$ 명령1 ; 명령2 ; 명령3 ; ....
    - Ex : \$ date ; who

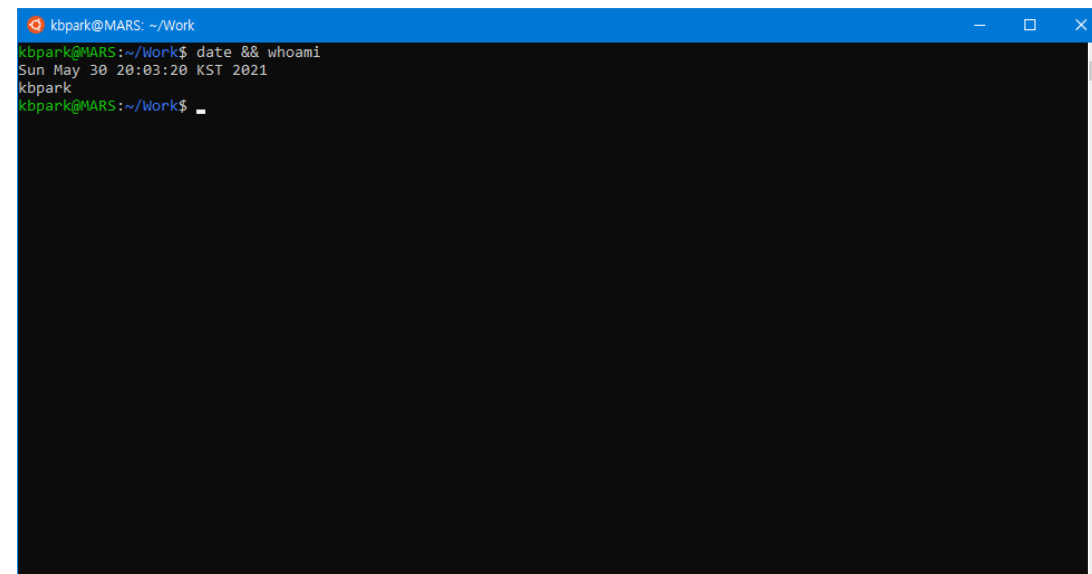
A terminal window titled 'kbpark@MARS: ~/Work' with a blue header bar. The terminal shows the command 'date ; whoami' being executed. The output is 'Sun May 30 20:00:47 KST 2021' followed by the username 'kbpark' on the next line. The prompt 'kbpark@MARS:~/Work\$' is visible at the bottom.

```
kbpark@MARS: ~/Work
kbpark@MARS:~/Work$ date ; whoami
Sun May 30 20:00:47 KST 2021
kbpark
kbpark@MARS:~/Work$
```

# LINUX - SHELL SCRIPT

## 여러 명령 사용

- 앰퍼샌드(&&)
  - 앞에서부터 순차적으로 실행하되, 명령 실행에 실패할 경우 뒤에 오는 명령은 실행하지 않는다.
  - \$ 명령1 && 명령2 && 명령3 && .....
  - Ex : \$ date && who
- Congratulation 여러분은 지금 쉘 스크립트를 작성 했다.
- 한 줄에 입력할 수 있는 명령어의 한계는 255자이다.
- 이렇게 스크립트를 실행 하려면 번번히 명령어 라인에 기입 해야 하는데 이것을 파일에 저장 해서 실행 해야 한다.

A terminal window titled 'kbpark@MARS: ~/Work' with a blue header bar. The terminal shows a sequence of commands and their outputs. The first command is 'date && whoami', which outputs 'Sun May 30 20:03:20 KST 2021' followed by the username 'kbpark'. The prompt then returns to 'kbpark@MARS:~/Work\$'.

```
kbpark@MARS: ~/Work
kbpark@MARS:~/Work$ date && whoami
Sun May 30 20:03:20 KST 2021
kbpark
kbpark@MARS:~/Work$
```

# LINUX - SHELL SCRIPT

## 셸 스크립트 파일 작성하기

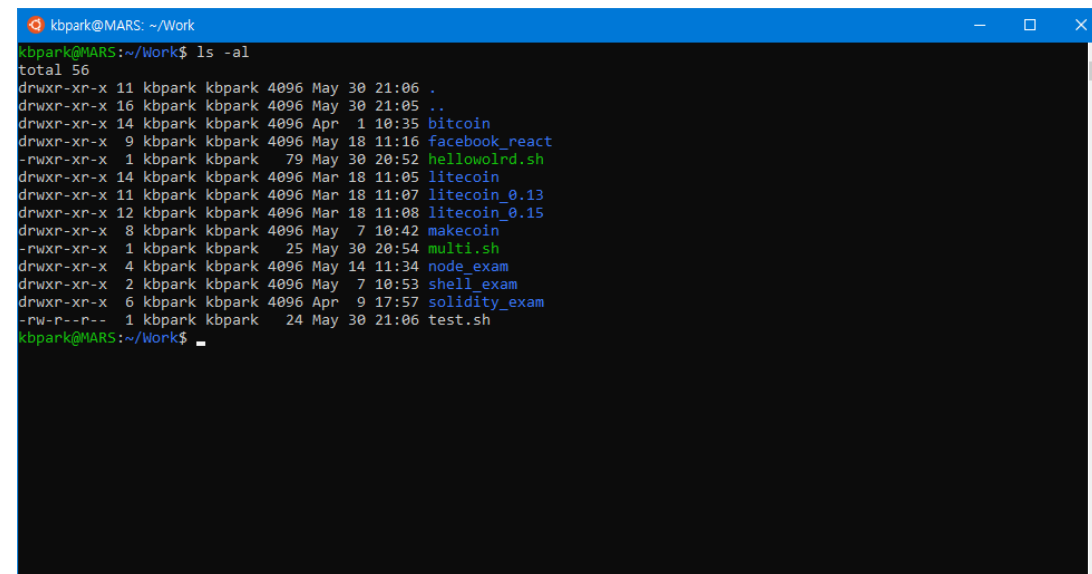
- 이제 첫 번째는 bash 셸이 스크립트 파일을 인식하는 것이다.
  - 셸은 명령을 찾기 위해 PATH라는 환경 변수에 입력하고 실행할 수 있도록 하는 것이다.
- 두 번째는 절대 경로에서 실행 하는 것이다.
  - `$ ./multicommand.sh`
  - `Bash: ./ multicommand.sh: Permission denied`

- `$ data ; whoami multicommand.sh` 스크립트
  - `#!/bin/bash`
  - `date`
  - `whoami`

# LINUX - SHELL SCRIPT

## 셸 스크립트 파일 작성하기

- 스크립트를 작성 했지만 보는 바와 같이 권한 에러가 난다.
  - 실행 권한이 없어 실행이 안되는 것이다.



```
kbpark@MARS: ~/Work
kbpark@MARS:~/Work$ ls -al
total 56
drwxr-xr-x 11 kbpark kbpark 4096 May 30 21:06 .
drwxr-xr-x 16 kbpark kbpark 4096 May 30 21:05 ..
drwxr-xr-x 14 kbpark kbpark 4096 Apr  1 10:35 bitcoin
drwxr-xr-x  9 kbpark kbpark 4096 May 18 11:16 facebook_react
-rwxr-xr-x  1 kbpark kbpark  79 May 30 20:52 hellowoird.sh
drwxr-xr-x 14 kbpark kbpark 4096 Mar 18 11:05 litecoin
drwxr-xr-x 11 kbpark kbpark 4096 Mar 18 11:07 litecoin_0.13
drwxr-xr-x 12 kbpark kbpark 4096 Mar 18 11:08 litecoin_0.15
drwxr-xr-x  8 kbpark kbpark 4096 May  7 10:42 makecoin
-rwxr-xr-x  1 kbpark kbpark  25 May 30 20:54 multi.sh
drwxr-xr-x  4 kbpark kbpark 4096 May 14 11:34 node_exam
drwxr-xr-x  2 kbpark kbpark 4096 May  7 10:53 shell_exam
drwxr-xr-x  6 kbpark kbpark 4096 Apr  9 17:57 solidity_exam
-rw-r--r--  1 kbpark kbpark  24 May 30 21:06 test.sh
kbpark@MARS:~/Work$
```

# LINUX - SHELL SCRIPT

## 셸 스크립트 파일 작성하기

- 실행 권한 주기
  - chmod
  - \$ chmod 755 multicommand.sh
  - .\$ /multicommand.sh

```
kbpark@MARS: ~/Work
drwxr-xr-x 14 kbpark kbpark 4096 Mar 18 11:05 litecoin
drwxr-xr-x 11 kbpark kbpark 4096 Mar 18 11:07 litecoin_0.13
drwxr-xr-x 12 kbpark kbpark 4096 Mar 18 11:08 litecoin_0.15
drwxr-xr-x 8 kbpark kbpark 4096 May 7 10:42 makecoin
-rwxr-xr-x 1 kbpark kbpark 25 May 30 20:54 multi.sh
drwxr-xr-x 4 kbpark kbpark 4096 May 14 11:34 node_exam
drwxr-xr-x 2 kbpark kbpark 4096 May 7 10:53 shell_exam
drwxr-xr-x 6 kbpark kbpark 4096 Apr 9 17:57 solidity_exam
-rw-r--r-- 1 kbpark kbpark 24 May 30 21:06 test.sh
kbpark@MARS:~/Work$ chmod 755 test.sh
kbpark@MARS:~/Work$ ls -al
total 56
drwxr-xr-x 11 kbpark kbpark 4096 May 30 21:06 .
drwxr-xr-x 16 kbpark kbpark 4096 May 30 21:05 ..
drwxr-xr-x 14 kbpark kbpark 4096 Apr 1 10:35 bitcoin
drwxr-xr-x 9 kbpark kbpark 4096 May 18 11:16 facebook_react
-rwxr-xr-x 1 kbpark kbpark 79 May 30 20:52 helloworld.sh
drwxr-xr-x 14 kbpark kbpark 4096 Mar 18 11:05 litecoin
drwxr-xr-x 11 kbpark kbpark 4096 Mar 18 11:07 litecoin_0.13
drwxr-xr-x 12 kbpark kbpark 4096 Mar 18 11:08 litecoin_0.15
drwxr-xr-x 8 kbpark kbpark 4096 May 7 10:42 makecoin
-rwxr-xr-x 1 kbpark kbpark 25 May 30 20:54 multi.sh
drwxr-xr-x 4 kbpark kbpark 4096 May 14 11:34 node_exam
drwxr-xr-x 2 kbpark kbpark 4096 May 7 10:53 shell_exam
drwxr-xr-x 6 kbpark kbpark 4096 Apr 9 17:57 solidity_exam
-rwxr-xr-x 1 kbpark kbpark 24 May 30 21:06 test.sh
kbpark@MARS:~/Work$ ./multi.sh
Sun May 30 21:13:38 KST 2021
kbpark
kbpark@MARS:~/Work$
```

# LINUX - SHELL SCRIPT

## 셸 스크립트 파일 작성하기

- 셸 스크립트는 셸에게 무슨 명령들을 실행할지 알려주는 스크립트 파일이다. 우리는 가장 널리 쓰이는 bash 셸을 사용하는 스크립트를 설명하도록 한다.
- `#!/bin/bash`
- 스크립트 최 상단에는 항상 이 구문이 적혀 있어야한다. 간단하게 hello, world라는 문자열을 출력하는 스크립트를 만들어봅시다. 파일명은 helloworld.sh로 한다.
  - `#!/bin/bash`
  - `echo "hello, world"`
  - `printf "hello, world"`

# LINUX - SHELL SCRIPT

## 메시지 표시하기

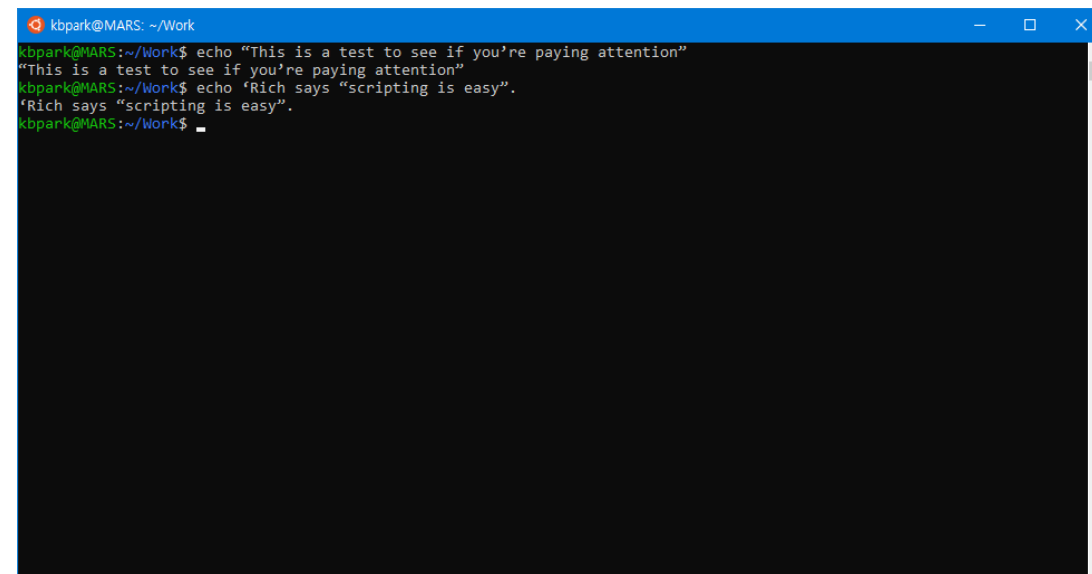
- 대부분의 셸 명령은 스크립트가 실행될 때 콘솔 모니터에 나름대로의 출력을 표시한다. 하지만 스크립트 사용자가 스크립트 안에서 무슨 일이 일어나고 있는지 알 수 있도록 별도의 문자 메시지를 추가하고 싶을 때가 자주 있을 것이다.
- echo 명령을 이용하면 이러한 일을 할 수 있다.
- echo 명령 뒤에 문자열을 추가 하면 텍스트 문자열을 표시할 수 있다.
  - `$ echo This is a test`
  - `This is a test`
  - `$`



# LINUX - SHELL SCRIPT

## 메시지 표시하기

- 표시하고자 하는 문자열을 나타내기 위해 따옴표(') 따로 사용하지 않아도 된다.
- 문자열 안에서 따옴표는 가끔 이상한 결과가 나온다.
  - `$ echo Let's see if this'll work`
  - `Lets see if thisll work`
- `echo` 명령은 텍스트 문자열을 묶기 위해서 홑따옴표 또는 겹따옴표를 쓴다. 문자열 안에서 따옴표를 사용하면 텍스트 안에서 한 가지 유형의 따옴표를 사용한 다음 문자열을 묶을 때에는 다른 유형의 따옴표를 써야 한다.
  - `$ echo "This is a test to see if you're paying attention"`
  - `This is a test to see if you're paying attention`
  - `$ echo 'Rich says "scripting is easy".'`
  - `Rich says "scripting is easy".`

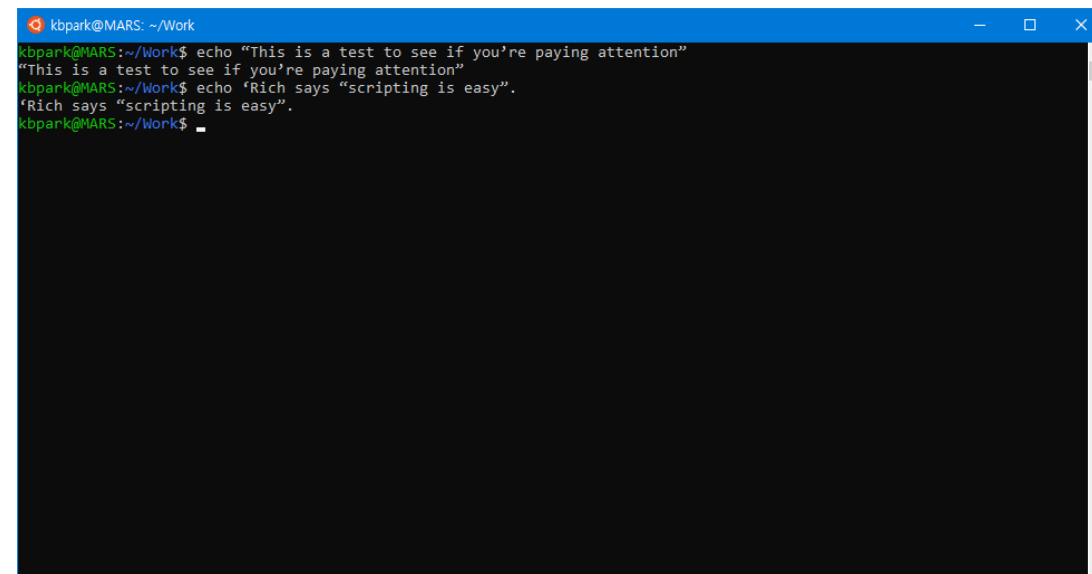


```
kbpark@MARS: ~/Work
kbpark@MARS:~/Work$ echo "This is a test to see if you're paying attention"
This is a test to see if you're paying attention
kbpark@MARS:~/Work$ echo 'Rich says "scripting is easy".'
Rich says "scripting is easy".
kbpark@MARS:~/Work$
```

# LINUX - SHELL SCRIPT

## 메시지 표시하기

- 표시하고자 하는 문자열을 나타내기 위해 따옴표(') 따로 사용하지 않아도 된다.
- 문자열 안에서 따옴표는 가끔 이상한 결과가 나온다.
  - `$ echo Let's see if this'll work`
  - `Lets see if thisll work`
- `echo` 명령은 텍스트 문자열을 묶기 위해서 홑따옴표 또는 겹따옴표를 쓴다. 문자열 안에서 따옴표를 사용하면 텍스트 안에서 한 가지 유형의 따옴표를 사용한 다음 문자열을 묶을 때에는 다른 유형의 따옴표를 써야 한다.
  - `$ echo "This is a test to see if you're paying attention"`
  - `This is a test to see if you're paying attention`
  - `$ echo 'Rich says "scripting is easy".'`
  - `Rich says "scripting is easy".`

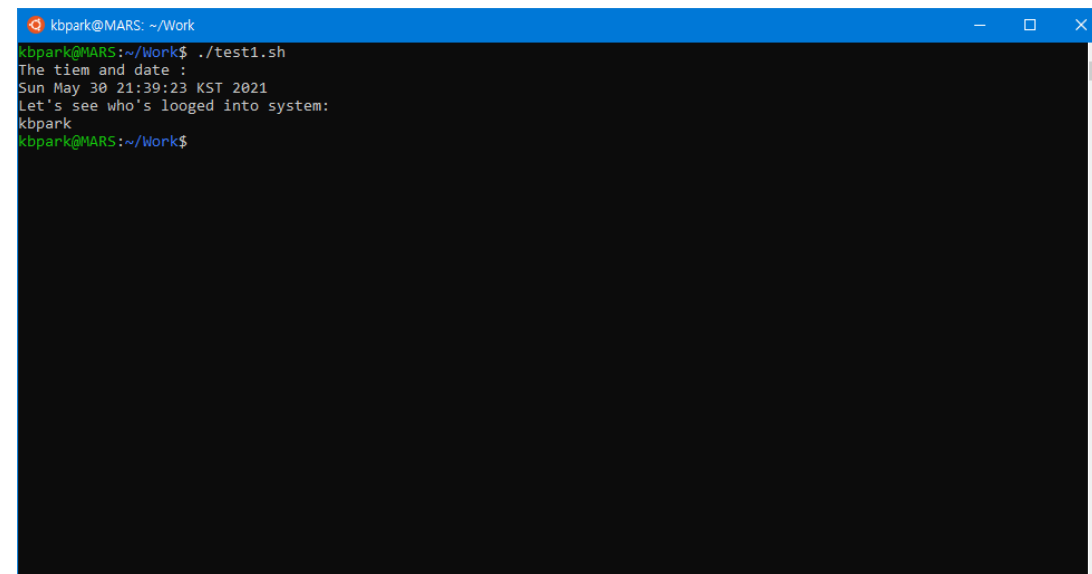


```
kbpark@MARS: ~/Work
kbpark@MARS:~/Work$ echo "This is a test to see if you're paying attention"
This is a test to see if you're paying attention
kbpark@MARS:~/Work$ echo 'Rich says "scripting is easy".'
Rich says "scripting is easy".
kbpark@MARS:~/Work$
```

# LINUX - SHELL SCRIPT

## 메시지 표시하기

- touch test1.sh
- vi test1.sh
  - #!/bin/bash/
  - #This is test shell script
  - echo The time and date are:
  - Date
  - echo "Let's see who's logged into the system:"
  - whoami

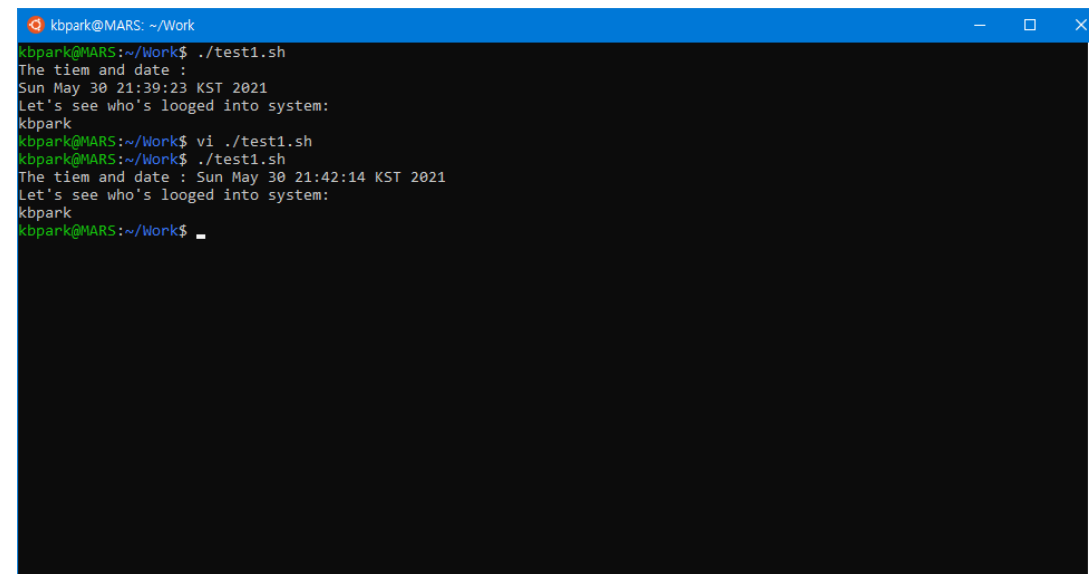


```
kbpark@MARS: ~/Work
kbpark@MARS:~/Work$ ./test1.sh
The time and date :
Sun May 30 21:39:23 KST 2021
Let's see who's logged into system:
kbpark
kbpark@MARS:~/Work$
```

# LINUX - SHELL SCRIPT

## 메시지 표시하기

- touch test1.sh
- vi test1.sh
  - #!/bin/bash/
  - #This is test shell script
  - echo -n "The time and date are:"
  - date
  - echo "Let's see who's logged into the system:"
  - whoami

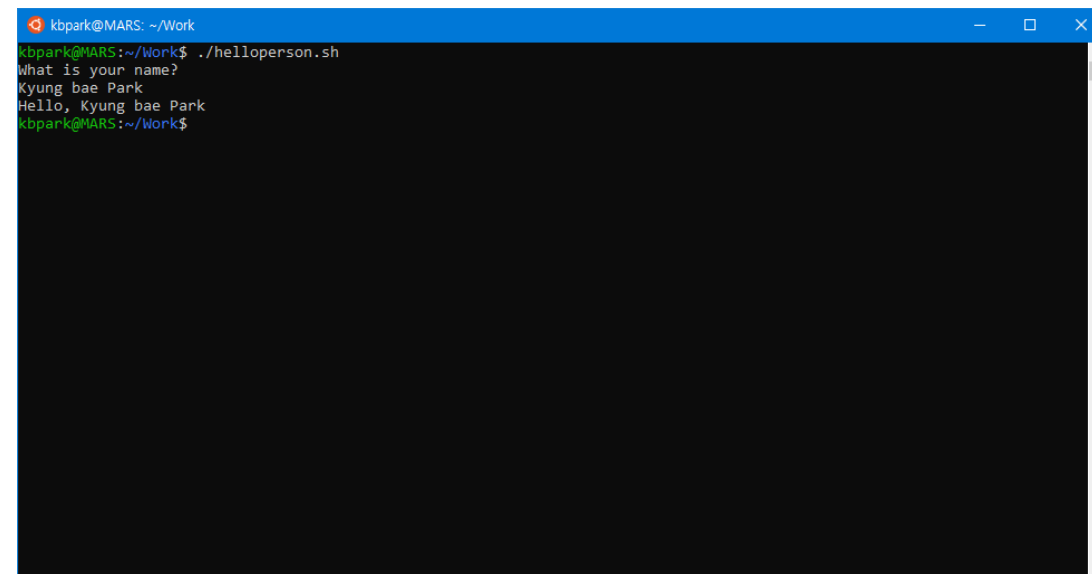


```
kbpark@MARS: ~/Work
kbpark@MARS:~/Work$ ./test1.sh
The tiem and date :
Sun May 30 21:39:23 KST 2021
Let's see who's looged into system:
kbpark
kbpark@MARS:~/Work$ vi ./test1.sh
kbpark@MARS:~/Work$ ./test1.sh
The tiem and date : Sun May 30 21:42:14 KST 2021
Let's see who's looged into system:
kbpark
kbpark@MARS:~/Work$
```

# LINUX - SHELL SCRIPT

## Extended shell script

- 셸 스크립트를 수행할 작업과 시기를 알려주는 필수 구성요소가 필요하다. 대부분의 셸 스크립트는 이보다 복잡하지만, 셸 스크립트 역시 일종의 프로그래밍 언어이며, 변수, 함수, 제어, 반복 등과 같은 구조로 이루어진다. 스크립트가 복잡한 구조로 되어 있어도 순차적인 실행 구조를 가진다.
- 다음은 간단한 입력 구조를 가지는 셸 스크립트 예제이다.
  - `#!/bin/bash`
  - `echo "What is your name?"`
  - `read PERSON`
  - `echo "Hello, $PERSON"`

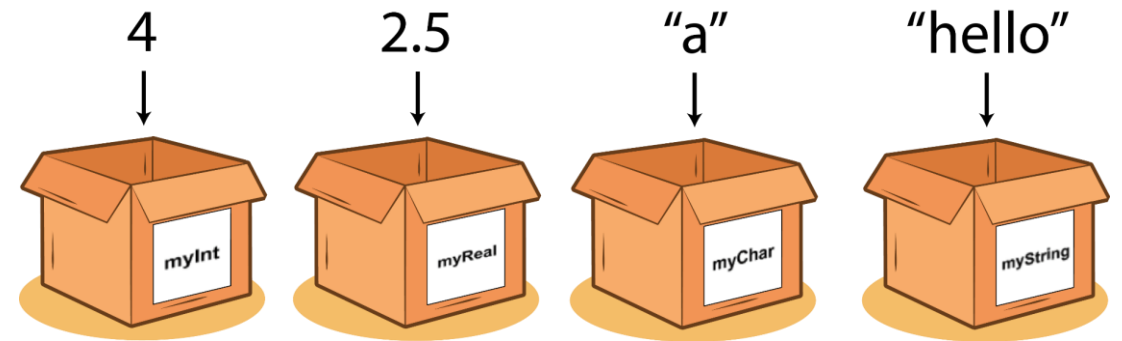
A terminal window titled 'kbpark@MARS: ~/Work' showing the execution of a script named 'helloperson.sh'. The script prompts 'What is your name?', the user enters 'Kyung bae Park', and the script outputs 'Hello, Kyung bae Park'. The prompt returns to 'kbpark@MARS:~/Work\$'.

```
kbpark@MARS: ~/Work
kbpark@MARS:~/Work$ ./helloperson.sh
What is your name?
Kyung bae Park
Hello, Kyung bae Park
kbpark@MARS:~/Work$
```

# LINUX - SHELL SCRIPT

## Using shell variable

- 셸에서 변수를 사용하는 방법에 대해 알아본다. 변수는 "아직 알려지지 않거나 어느 정도까지만 알려져 있는 양이나 정보에 대한 상징적인 이름이다. 컴퓨터 소스 코드에서의 변수 이름은 일반적으로 데이터 저장 위치와 그 안의 내용물과 관련되어 있으며 이러한 것들은 프로그램 실행 도중에 변경될 수 있다."
- 이를 쉽게 정의하면 다음과 같다.
  - 변수는 컴퓨터 메모리에 존재한다.
  - 할당된 메모리 공간은 정보를 저장하기 위해서 사용된다.
  - 정보가 저장된 공간을 찾기 위해서, 이름을 붙여서 사용한다.
- 변수의 할당된 값은 숫자, 텍스트 파일, 파일 이름, 장치 또는 다른 유형의 데이터일 수 있으며, 변수는 할당된 메모리의 주소를 나타내는 포인터이기 때문에 변수를 생성, 할당, 삭제가 가능하다.



# LINUX - SHELL SCRIPT

## Variable Names

- 셸에서 변수 이름을 지칭하는 규칙은 다음과 같다.
  - 변수 안에 들어갈 수 있는 글자는 a to z, A to Z이다.
  - 변수 안에 들어갈 수 있는 숫자는 0 ~ 9까지 이다.
  - 서로 다른 변수 이름을 이어서 사용하기 원한다면 underscore character ( \_ )을 사용한다.
  - 셸 변수의 이름은 대문자를 사용한다.

# LINUX - SHELL SCRIPT

## Variable Names

- 올바른 변수 선언의 예제를 살펴보면 다음과 같다.
  - \_ALL
  - NAME
  - VAR\_1
  - VAR\_2
- 잘못된 변수 선언의 예제는 다음과 같다.
  - 2\_VAR
  - -VARIABLE
  - VAR1-VAR2
  - VAR\_A!

**셸에서!, -, \*와 같은 특수문자를 사용할 수 없는 이유는 셸 자체에서 지칭하는 의미가 존재하기 때문이다.**



# LINUX - SHELL SCRIPT

## Defining Variables

- 변수를 정의하는 일반적인 방법은 다음과 같다.
  - `variable_name=variable_value`
  - `NAME="Lucas"`
- 예제에서 확인할 수 있듯이 NAME이라는 변수를 정의하고 "Lucas"라는 값을 대입할 수 있다. 이러한 유형의 변수를 **스칼라** 변수라고 지칭하는데 스칼라 변수는 한 번에 하나의 값을 저장할 수 있다는 특징을 가지고 있다.
  - `ORGANIZATIONS="wisoft"`
  - `NUMER_OF_PEOPLE=30`

# LINUX - SHELL SCRIPT

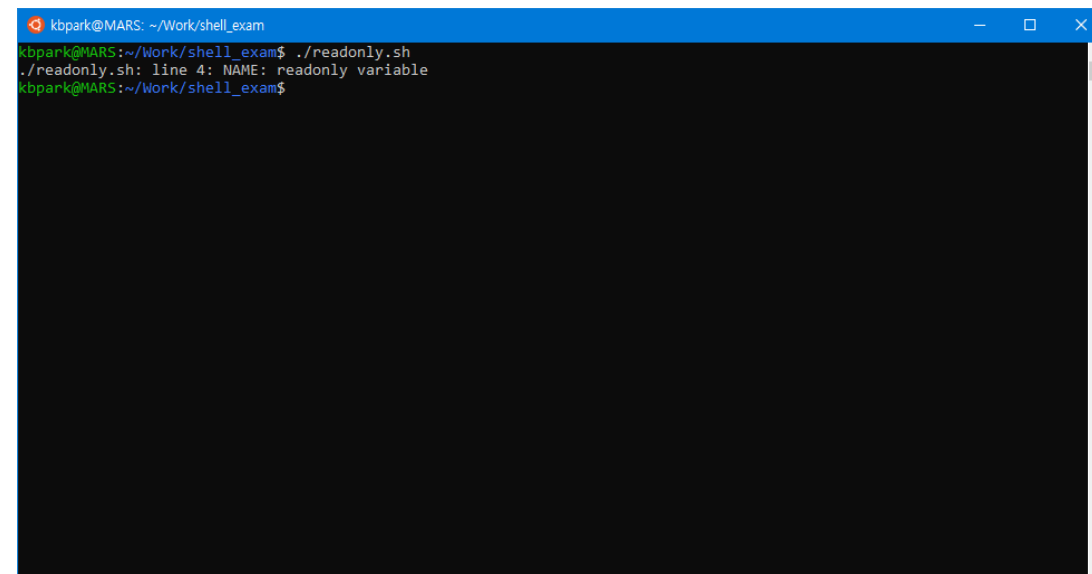
## Accessing Values

- 변수에 저장된 값에 접근하기 위해서는 이름 앞에 \$기호를 붙여야 한다. 다음 예제는 정의된 변수 NAME 값에 접근하고 **STDOUT**으로 출력한다.
  - `#!/bin/bash`
  - `NAME="Lucas"`
  - `echo $NAME`

# LINUX - SHELL SCRIPT

## Read-only Variables

- 셸에서는 읽기 전용으로 변수를 지정할 수 있으며, 읽기 전용으로 지정된 변수는 값을 변경할 수 없다.
- 다음 예제는 읽기 전용 변수에서 값을 변경하고자 할 때 나타나는 에러를 보여준다.
  - `#!/bin/bash`
  - `NAME="Lucas"`
  - `readonly NAME`
  - `NAME="Kyung Bae Park"`
- `$ ./readonly.sh: line 4: NAME: readonly variable`

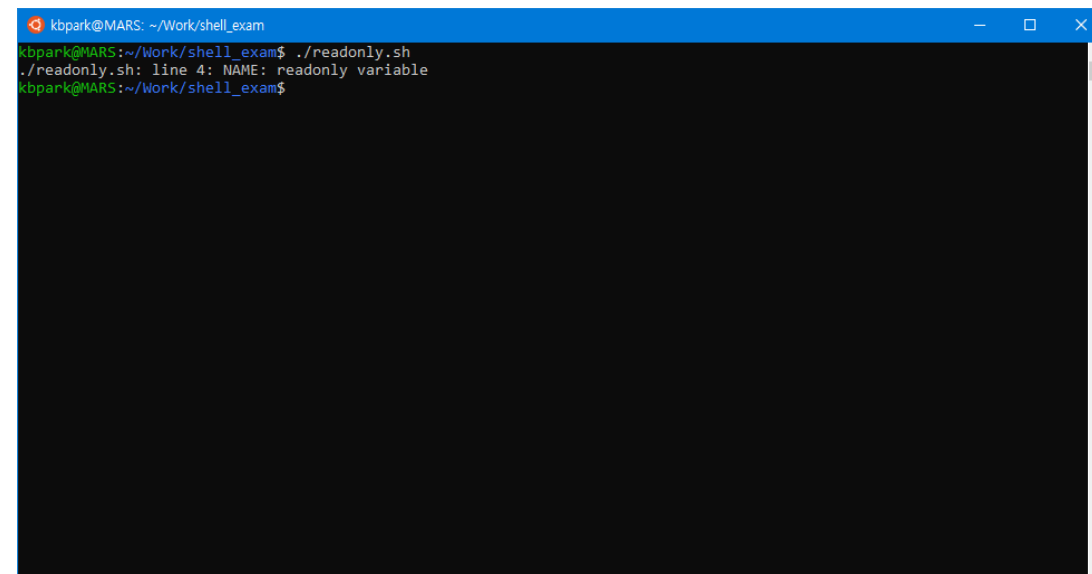
A terminal window titled 'kbpark@MARS: ~/Work/shell\_exam' with a blue header bar. The terminal shows the execution of a script named 'readonly.sh'. The first line is 'kbpark@MARS:~/Work/shell\_exam\$ ./readonly.sh'. The second line shows the output of the script: './readonly.sh: line 4: NAME: readonly variable'. The prompt returns to 'kbpark@MARS:~/Work/shell\_exam\$'.

```
kbpark@MARS: ~/Work/shell_exam
kbpark@MARS:~/Work/shell_exam$ ./readonly.sh
./readonly.sh: line 4: NAME: readonly variable
kbpark@MARS:~/Work/shell_exam$
```

# LINUX - SHELL SCRIPT

## Unsetting Variables

- 변수에 할당된 값을 해제하면 더 이상 변수에 접근할 수 없다.
  - `#!/bin/bash`
  - `NAME="Lucas"`
  - `unset NAME`
  - `echo $NAME`
- 위의 변수를 실행하게 되면 어떠한 것도 출력되지 않으며, 읽기 전용으로 선언된 변수는 `unset` 할 수 없다.



```
kbpark@MARS: ~/Work/shell_exam
kbpark@MARS:~/Work/shell_exam$ ./readonly.sh
./readonly.sh: line 4: NAME: readonly variable
kbpark@MARS:~/Work/shell_exam$
```

# LINUX - SHELL SCRIPT

## Variable Types

- 셸이 실행하기 위해서는 다음과 같은 3가지의 주요 변수가 존재한다.
- 지역변수
  - 셸의 인스턴스에 존재하는 변수로 기본적으로 셸에서 지정한 모든 변수는 전역 변수로 선언되기 때문에 지역변수를 사용하기 위해서는 local 키워드를 반드시 사용해야 한다.
  - 혹은 셸을 실행할 때 인자 값으로 넘겨줄 수 있다.
- 환경변수
  - 셸 스크립트를 통해 작성된 프로그램 중에서 정상적으로 동작하기 위한 변수이다.
  - 선언된 변수는 모든 자식 프로세스에서 접근하여 사용할 수 있으며, 프로그램이 실행하기 위해 참조되는 경우 외에는 사용하면 안 된다.

# LINUX - SHELL SCRIPT

## Variable Types

### ■ 셸 변수

- 셸이 동작하기 위해 필요한 특수 변수이다.
- set 명령을 통해 셸 변수를 확인할 수 있다.
- 우리가 흔히 알고 있는 셸 변수로는 \$PATH, \$HOME 등이 있다.

```
$ set
'!'=0
'#'=0
'$'=985
'*'=( )
-=569JNRXZghiklms
0=-zsh
'?'=0
@=( )
ARGC=0
BG
CDPATH=''
COLORFGBG='7;0'
COLORTERM=truecolor
COLUMNS=114
COMMAND_MODE=unix2003
COMP_WORDBREAKS=:
CPUTYPE=x86_64
CURRENT_BG=NONE
```

# LINUX - SHELL SCRIPT - SHELL 특수 변수

## 예약 변수 (Reserved Variable)

- 일반적인 프로그래밍 언어에서 사용하는 예약 변수와 동일한 기능을 담당한다고 생각하면 된다. 쉘 프로그래밍을 작성할 때, 예약 변수를 사용하면 보편적인 실행환경으로 작성할 수 있으므로 편리하게 사용할 수 있다.

# LINUX - SHELL SCRIPT - SHELL 특수 변수

## 예약 변수 (Reserved Variable)

변수명	설명
HOME	사용자 홈 디렉터리
PATH	실행 파일을 찾을 경로
LANG	프로그램 사용시 기본 지원되는 언어
FUNCNAME	현재 함수 이름
SECONDS	스크립트가 실행된 시간 (초 단위)
SHLVL	중첩된 셸 레벨 (현재 실행중인 셸 레벨 수이며, Linux 배포판에 따라 다름)
SHELL	로그인해서 사용하고 있는 셸
PPID	부모 프로세스의 PID
BASH	BASH 실행 파일 경로
BASH_ENV	스크립트 실행시 BASH 시작 파일을 읽을 위치 변수

변수명	설명
BASH_VERSION	현재 설치된 BASH 버전
BASH_VERSINFO	배열로 상세 정보 출력
MAIL	메일 보관 경로
MAILCHECK	메일 확인 시간
OSTYPE	운영체제 종류
TERM	터미널의 종류
HOSTNAME	호스트 이름
HOSTTYPE	시스템 하드웨어 종류
MACHTYPE	머신 종류
LOGNAME	로그인 이름



# LINUX - SHELL SCRIPT - SHELL 특수 변수

## 예약 변수 (Reserved Variable)

변수명	설명
UID	사용자 UID
TMOUNT	로그아웃할 시간, 0일 경우 무제한
USER	사용자의 이름
USERNAME	사용자 이름
GROUPS	사용자 그룹 (/etc/passwd)
HISTFILE	히스토리 파일 경로
HISTFILESIZE	히스토리 파일 사이즈
HISTSIZE	히스토리에 저장된 개수
HISTCONTROL	중복되는 명령에 대한 기록 유무
DISPLAY	X 디스플레이 이름
IFS	입력 필드 구분자

변수명	설명
VISAUL	VISUAL 편집기 이름
EDITOR	기본 편집기 이름
COLUMNS	터미널의 컬럼 수
LINES	터미널의 라인 수
LS_COLORS	ls 명령의 색상 관련 옵션
PS1	기본 프롬프트 스트링. 기본값은 <code>`\s-\v\$ '</code>
PS2	긴 문자 입력을 위해 나타나는 문자열. 기본 값은 <code>&gt;</code>
PS3	셸 스크립트에서 select 사용시 프롬프트 변수(기본 값: <code>#?</code> )
PS4	셸 스크립트 디버깅 모드의 프롬프트 변수

# LINUX - SHELL SCRIPT

## 특수 권한

- 일반적으로 리눅스 유닉스는 사용자의 파일 권한을 부여하여 기초적인 보안체계를 유지한다. 파일이나 디렉터리에서는 user, group, other 권한이 존재하며 각각 읽기, 쓰기, 실행 권한을 부여할 수 있다.
- 이러한 권한을 수정하기 위해서 chmod 명령을 사용하는데 셸에서 특수 권한을 주기 위해서도 같은 명령을 사용한다.

# LINUX - SHELL SCRIPT

## SetUID

- 파일을 실행할 때 일시적으로 소유자의 권한을 얻어 실행할 수 있도록 한다.
- root 권한으로 지정된 프로그램에 SetUID가 지정되어 있으면 root 권한으로 실행된다.
- 기존에 실행 권한이 없으면 대문자 S, 있으면 소문자 s로 표시된다.

```
$ touch setuid
$ ll
-rw-r--r-- 1 seongwon staff 0B 1 25 14:50 setuid

$ chmod 4644 ./setuid
-rwsr--r-- 1 seongwon staff 0B 1 25 14:50 setuid
```

# LINUX - SHELL SCRIPT

## SetUID

- 리눅스에서 설정되어 있는 대표적인 파일은 /usr/bin/passwd이다.
  - 해당 파일은 계정의 비밀번호를 변경할 수 있도록 하는 실행파일로 root만 변경 가능하도록 설정되어 있다.
  - /usr/bin/passwd에 SetUID가 설정되어 있지 않으면 일반 사용자는 root 사용자를 통해 비밀번호를 변경해야 하므로 일반 사용자도 /etc/passwd 파일을 수정 가능하도록 설정되어 있다.
  - /etc/passwd는 사용자의 이름과 같은 권한에 대한 데이터 베이스만 존재하며, 실제 비밀번호는 /usr/bin/passwd 사용자가 패스워드를 변경할 때만 사용한다. 과거 패스워드가 파일 형태로 저장되어 있으나 현재는 저장되지 않는다. 이러한 부분에 대한 자세한 설명은 shadow파일의 대해 참조하길 바란다.

```
$ ls -l /usr/bin/passwd  
-rwsr-xr-x 1 root root 68208 May 28 2020 /usr/bin/passwd
```

# LINUX - SHELL SCRIPT

## SetGID

- 파일을 실행할 때 일시적으로 파일 소유 그룹의 권한을 얻어 실행할 수 있도록 한다.
  - 기존 권한에 실행 권한이 없으면 대문자 S, 있으면 소문자 s로 표시된다.

```
$ touch setgid
$ ll
-rw-r--r-- 1 seongwon staff 0B 1 25 15:20 setgid

$ chmod 2644 ./setuid
-rw-r-Sr-- 1 seongwon staff 0B 1 25 15:20 setgid
```

# LINUX - SHELL SCRIPT

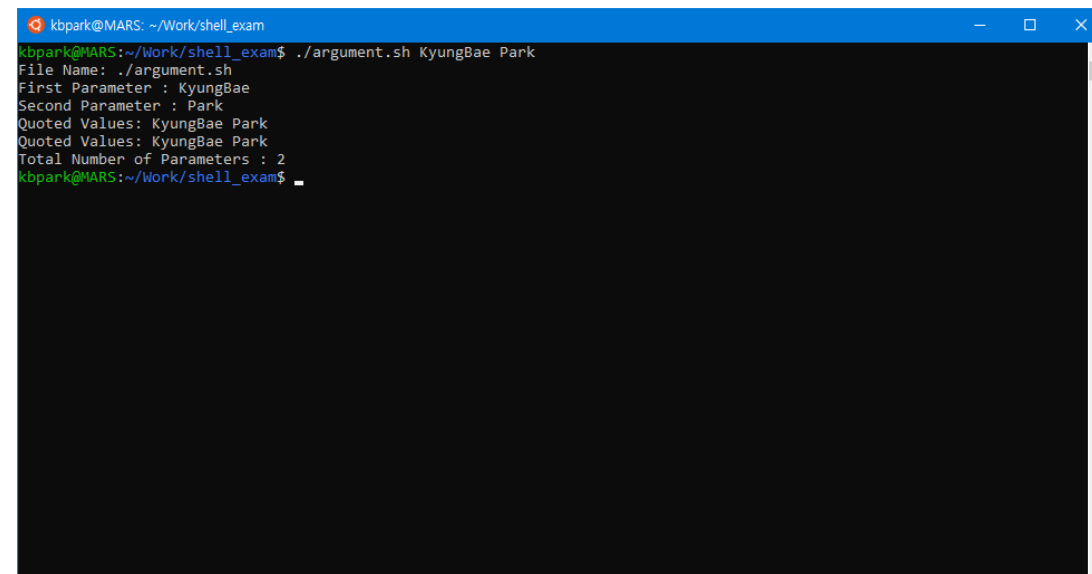
## 특수 변수

\$0	현재 스크립트 파일 이름
\$n	스크립트가 호출된 인수, n은 인수의 십진수를 표현 (\$1, \$2, \$3 ...)
\$#	매개 변수의 총 개수
\$*	전체 인자 값
\$@	모든 인자가 ""로 묶여 있으며, 스크립트가 두 개의 인수를 받으면 \$1, \$2와 동일하다.
\$?	마지막으로 실행된 명령어, 함수, 스크립트 자식의 종료 상태
\$\$	현재 스크립트의 PID
#!	마지막으로 실행된 백그라운드 PID

# LINUX - SHELL SCRIPT

## Command-Line Arguments

- 위에서 설명한 특수 변수를 활용하여 쉘 스크립트를 작성하고, 인수를 넘긴다.
  - `#!/bin/bash`
  - `echo "File Name: $0"`
  - `echo "First Parameter : $1"`
  - `echo "Second Parameter : $2"`
  - `echo "Quoted Values: $@"`
  - `echo "Quoted Values: $*"`
  - `echo "Total Number of Parameters : $#"`

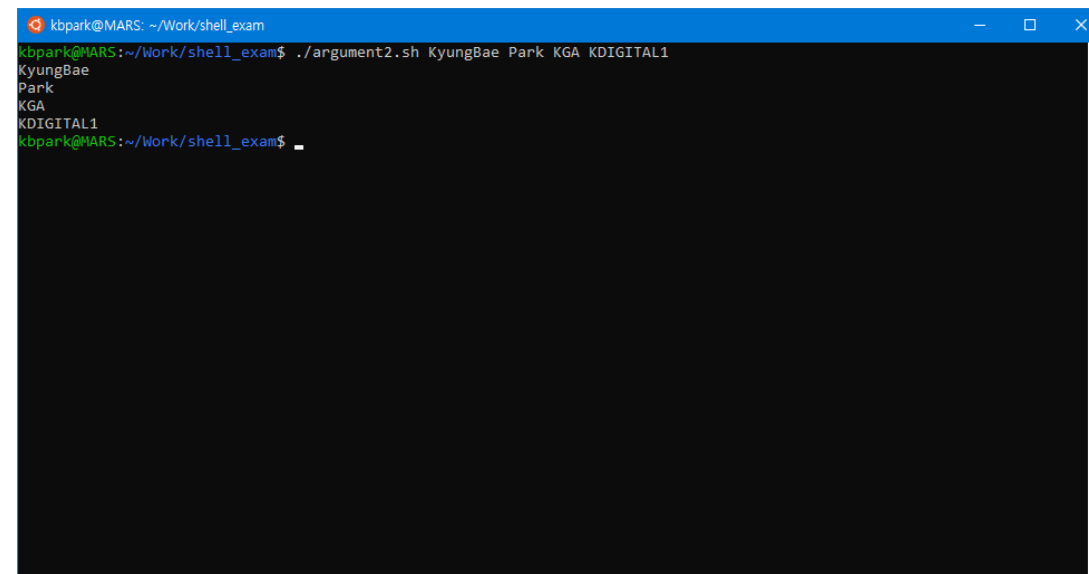


```
kbpark@MARS: ~/Work/shell_exam
kbpark@MARS:~/Work/shell_exam$ ./argument.sh KyungBae Park
File Name: ./argument.sh
First Parameter : KyungBae
Second Parameter : Park
Quoted Values: KyungBae Park
Quoted Values: KyungBae Park
Total Number of Parameters : 2
kbpark@MARS:~/Work/shell_exam$
```

# LINUX - SHELL SCRIPT

## Command-Line Arguments

- `#!/bin/bash`
- `for TOKEN in $*`
- `do`
- `echo $TOKEN`
- `done`



```
kbpark@MARS: ~/Work/shell_exam
kbpark@MARS:~/Work/shell_exam$ ./argument2.sh KyungBae Park KGA KDIGITAL1
KyungBae
Park
KGA
KDIGITAL1
kbpark@MARS:~/Work/shell_exam$
```



## 과제 - C++

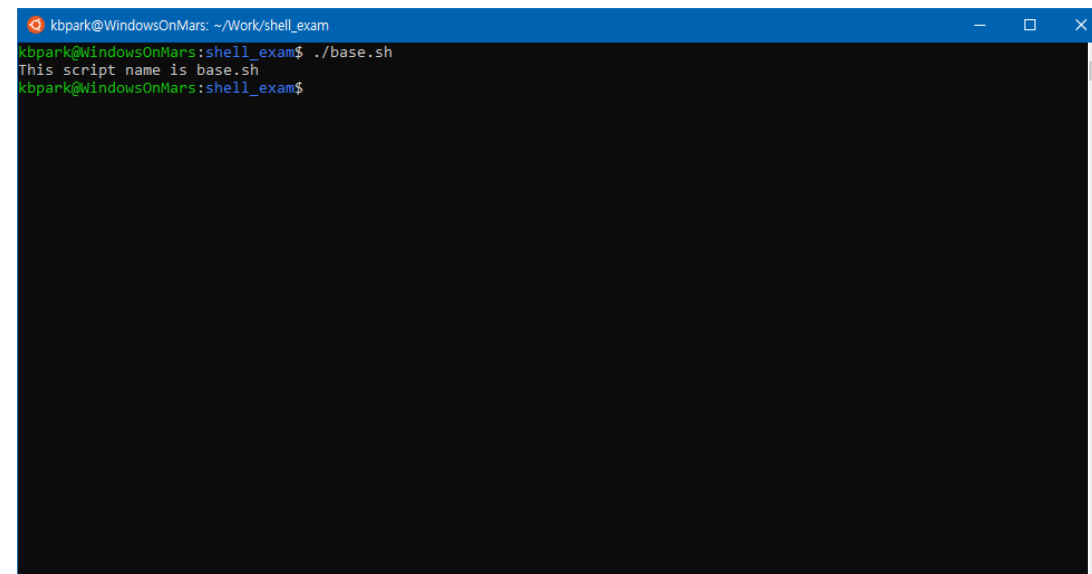
삼각형을 나타내는 객체를 만들기 위한 클래스를 선언하려고 한다. 삼각형은 세 개의 꼭짓점 좌표 ( $x[0]$ ,  $y[0]$ ), ( $x[1]$ ,  $y[1]$ ), ( $x[2]$ ,  $y[2]$ )로 정의된다. 삼각형 객체는 다음과 같은 행동을 할 수 있다.

- 세 개의 꼭짓점 좌표(6개의 double형 값)를 매개 변수로 받아 삼각형 객체를 생성한다(생성자).
- 삼각형을 x축으로 dx, y축으로 dy만큼 이동할 수 있다.  
(예) `t.move(dx, dy);` 와 같이 호출하면 삼각형 객체 t의 모든 꼭짓점 좌표들을 모두 (dx, dy)만큼 이동함
- 삼각형을 x축 방향으로 sx배, y축 방향으로 sy배 크기조정을 할 수 있다.  
(예) `t.scale(sx, sy);` 와 같이 호출하면 삼각형 객체 t의 모든 꼭짓점의 x좌표를 sx배, y좌표를 sy배 크기조정한다.
- 삼각형의 면적을 구할 수 있다.  
(참고) 삼각형의 면적은 다음과 같이 구할 수 있다.  
면적 =  $| (x[0] \times y[1] - y[0] \times x[1]) + (x[1] \times y[2] - y[1] \times x[2]) + (x[2] \times y[0] - y[2] \times x[0]) | / 2$

# LINUX - SHELL SCRIPT

## 명령어 치환(Command Substitution)

- 명령어 치환은 하나나 그 이상의 명령어의 출력을 재할당 해준다. 명령어 치환은 말그대로 한 명령어의 출력을 다른 문맥으로 연결해 준다.
- 명령어 치환의 두가지 형태
  - 역따옴표(`...`)를 쓰는 것이다
  - `$(...)` 양식
  - `#!/bin/bash`
- `script_name=`basename $0``
- `echo "This script name is $script_name"`



```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ ./base.sh
This script name is base.sh
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## 명령어 치환(Command Substitution)

- 명령어 치환은 Bash 에서 쓸 수 있는 툴셋을 확장시켜 줄 수 있다.
- 표준출력으로 결과를 출력해 주는(잘 동작하는 유닉스 툴이 그래야 하는 것처럼) 프로그램이나 스크립트를 짜고 그 결과를 변수로 할당하면 된다.

```
#include <stdio.h>
```

```
using namespace std;
```

```
/* "Hello, world." C++ program */
```

```
int main()
```

```
{
```

```
    cout << "Hello, world." endl;
```

```
    return (0);
```

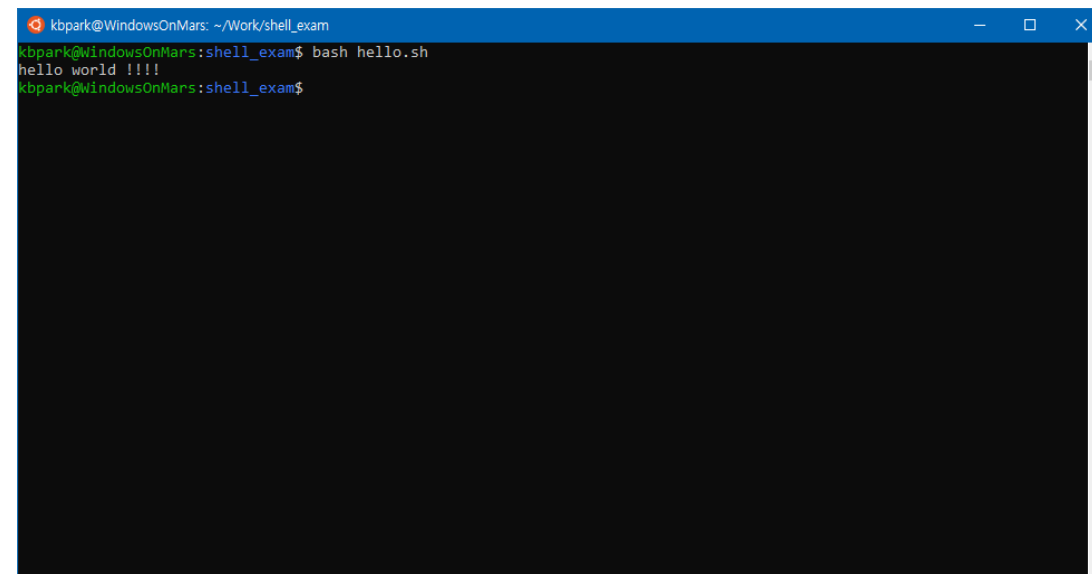
```
}
```

```
$ g++ -o hello hello.cpp
```

# LINUX - SHELL SCRIPT

## 명령어 치환(Command Substitution)

- `#!/bin/bash`
- `# hello.sh`
- `greeting=`./hello``
- `echo $greeting`



```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash hello.sh
hello world !!!!
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## 입력과 출력 리다이렉트

- 때로는 모니터에 표시된 명령의 출력을 저장할 필요가 있다 bash 셸은 명령의 출력을 다른 위치에(like a file) 리다이렉트할 수 있도록 몇 가지 연산자를 제공한다.
- 리다이렉트는 출력은 물론 입력에도 사용될 수 있으며 파일을 명령에 입력시킬수있다.

# LINUX - SHELL SCRIPT

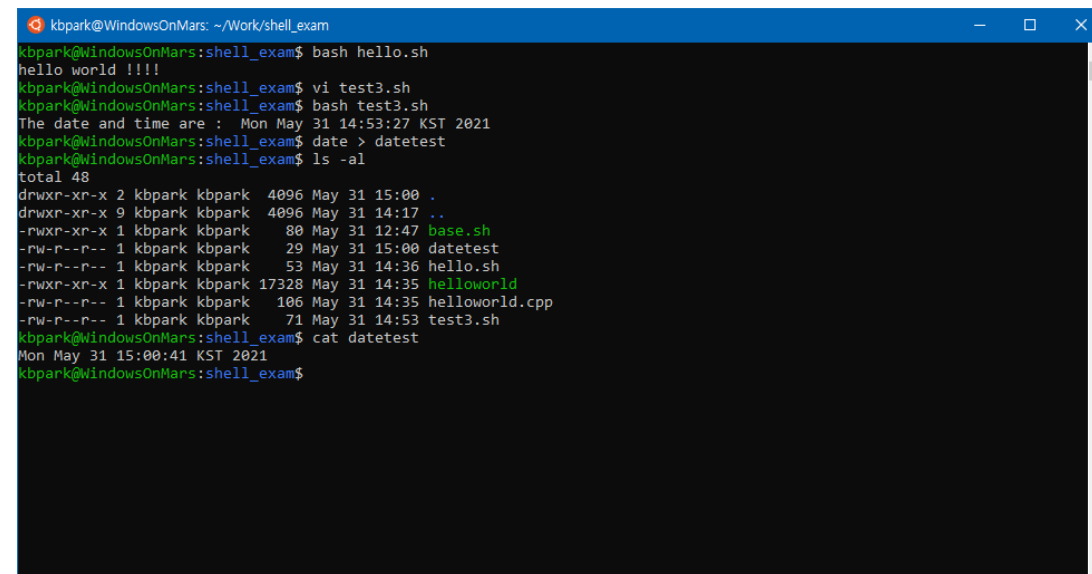
## 출력 리다이렉트

- 리다이렉트의 가장 기본적인 형태는 파일 명령의 출력을 전송한다.
- Bash 셸은 이를 위해 > 부등호 기호를 상용한다.
  - `$ command > outfile`
- 출력 파일에 저장되는 모니터에 표시할 수 있는 모든 명령의 출력은 지정된 파일에 대신 저장 된다.
  - `date > datetest`

# LINUX - SHELL SCRIPT

## 출력 리다이렉트

- 출력 파일에 저장되는 모니터에 표시할 수 있는 모든 명령의 출력은 지정된 파일에 대신 저장 된다.
  - `date > datetest`

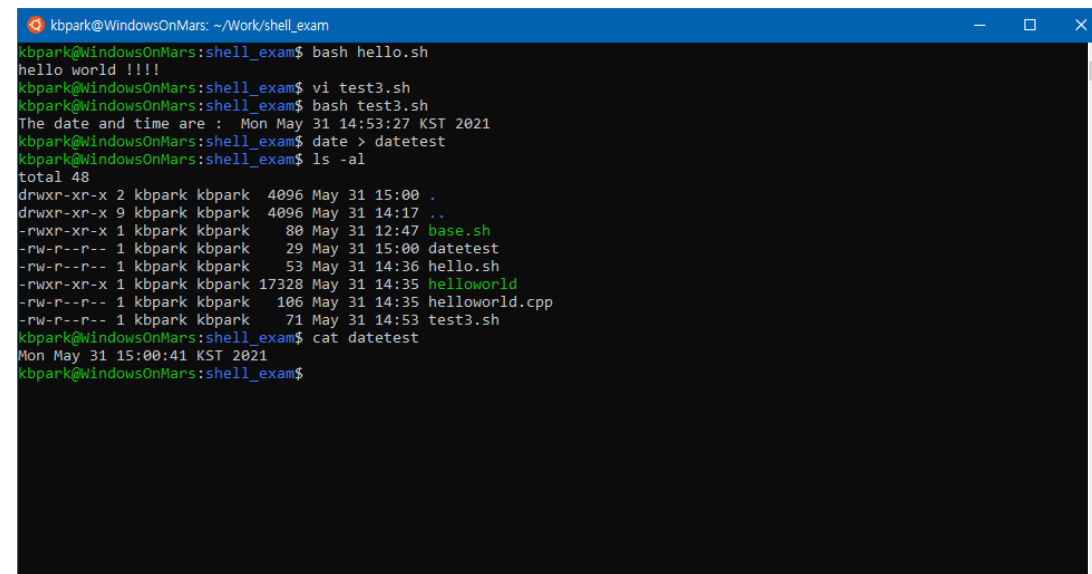


```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash hello.sh
hello world !!!!
kbpark@WindowsOnMars:shell_exam$ vi test3.sh
kbpark@WindowsOnMars:shell_exam$ bash test3.sh
The date and time are : Mon May 31 14:53:27 KST 2021
kbpark@WindowsOnMars:shell_exam$ date > datetest
kbpark@WindowsOnMars:shell_exam$ ls -al
total 48
drwxr-xr-x 2 kbpark kbpark 4096 May 31 15:00 .
drwxr-xr-x 9 kbpark kbpark 4096 May 31 14:17 ..
-rwxr-xr-x 1 kbpark kbpark 80 May 31 12:47 base.sh
-rw-r--r-- 1 kbpark kbpark 29 May 31 15:00 datetest
-rw-r--r-- 1 kbpark kbpark 53 May 31 14:36 hello.sh
-rwxr-xr-x 1 kbpark kbpark 17328 May 31 14:35 helloworld
-rw-r--r-- 1 kbpark kbpark 106 May 31 14:35 helloworld.cpp
-rw-r--r-- 1 kbpark kbpark 71 May 31 14:53 test3.sh
kbpark@WindowsOnMars:shell_exam$ cat datetest
Mon May 31 15:00:41 KST 2021
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## 출력 리다이렉트

- > 리다이렉트 연산자는 파일 datetest를 만들고 date 명령의 출력을 저장 했다.
- 출력 파일이 이미 존재하면 리다이렉트는 > 연산자는 기존의 파일을 새로운 파일데이터로 덮어 씩운다.
  - \$ whoami > datetest



```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash hello.sh
hello world !!!!
kbpark@WindowsOnMars:shell_exam$ vi test3.sh
kbpark@WindowsOnMars:shell_exam$ bash test3.sh
The date and time are : Mon May 31 14:53:27 KST 2021
kbpark@WindowsOnMars:shell_exam$ date > datetest
kbpark@WindowsOnMars:shell_exam$ ls -al
total 48
drwxr-xr-x 2 kbpark kbpark 4096 May 31 15:00 .
drwxr-xr-x 9 kbpark kbpark 4096 May 31 14:17 ..
-rwxr-xr-x 1 kbpark kbpark 80 May 31 12:47 base.sh
-rw-r--r-- 1 kbpark kbpark 29 May 31 15:00 datetest
-rw-r--r-- 1 kbpark kbpark 53 May 31 14:36 hello.sh
-rwxr-xr-x 1 kbpark kbpark 17328 May 31 14:35 helloworld
-rw-r--r-- 1 kbpark kbpark 106 May 31 14:35 helloworld.cpp
-rw-r--r-- 1 kbpark kbpark 71 May 31 14:53 test3.sh
kbpark@WindowsOnMars:shell_exam$ cat datetest
Mon May 31 15:00:41 KST 2021
kbpark@WindowsOnMars:shell_exam$
```



# LINUX - SHELL SCRIPT

## 입력 리다이렉트

- 입력 리다이렉트는 출력과는 반대다.
- 입력 리다이렉트 파일은 파일의 내용을 받아서 명령으로 보낸다.
- 입력 리다이렉트 기호는 < 부등호 이다.
  - `$ command < infile`
- 기억하기 쉬운 방법은 언제나 커맨드라인의 처음에 나온다는 것이고, 리다이렉트 기호는 부등호가 아니라 데이터의 흐르는 방향을 '가리키는' 기호다.
- < 기호는 데이터가 입력 파일로부터 명령으로 흐르는 것을 나타낸다.
  - `$ wc < testinput`
- Wc 명령은 데이터의 텍스트 양을 계산한다. 기본적으로 3가지 값을 나타낸다.
  - 텍스트의 줄 수
  - 텍스트의 단어 수
  - 텍스트의 바이트 수

# LINUX - SHELL SCRIPT

## 배열

- 우리가 지금까지 쉘 스크립트에서 변수를 선언할 때 다음과 같은 방법을 사용했다.
  - `#!/bin/bash`
  - `NAME01="Lucas"`
  - `NAME02="KyungBae"`
  - `NAME03="KGA"`
  - `NAME04="KDIGITAL"`
  - `echo $NAME01`
- 옆에서 정의한 변수를 배열로 정의하면 다음과 같이 사용할 수 있다.
  - `#!/bin/bash`
  - `NAME[0]="Lucas"`
  - `NAME[1]=" KyungBae"`
  - `NAME[2]=" KGA"`
  - `NAME[3]=" KDIGITAL"`
  - `echo "First Index: ${NAME[0]}"`
  - `echo "Second Index: ${NAME[1]}"`

# LINUX - SHELL SCRIPT - 배열

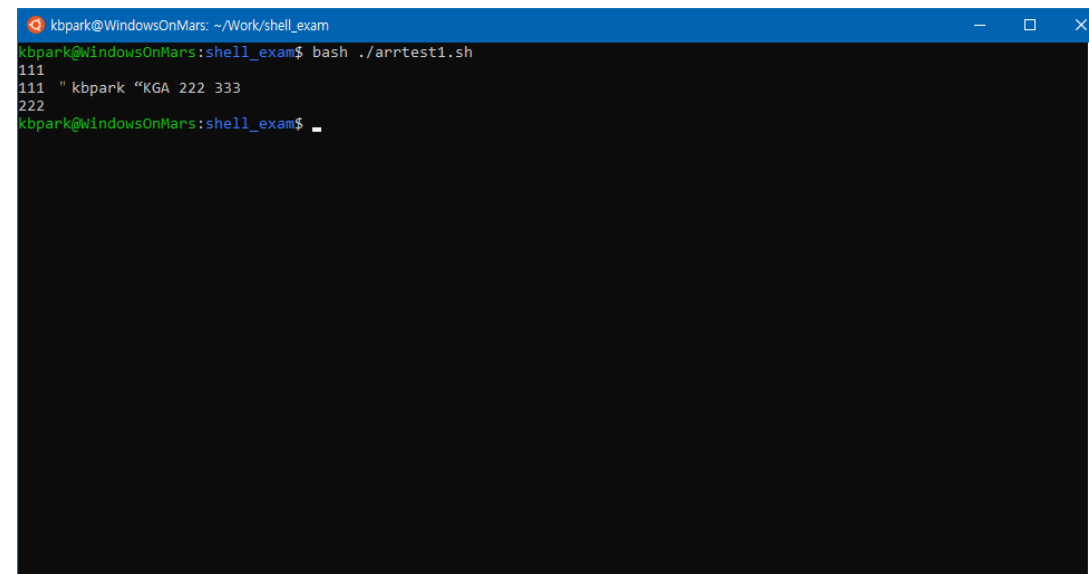
## 배열

- 만약 모든 배열에 접근하고자 한다면 다음 방법으로 수행할 수 있다.
  - `${array_name[*]}`
  - `${array_name[@]}`
- 셸 스크립트에서 배열을 정의하는 방법은 다음과 같다.
  - `array_name=("element 1" "element 2" "element 3")`

# LINUX - SHELL SCRIPT - 배열

## 배열

- 다음 예제를 수행하여 결과를 확인한다.
  - `#!/bin/bash`
  - `array=(111 " kbpark" "KGA" 222 333)`
  - `echo $array`
  - `echo ${array[*]}`
  - `echo ${array[2]}`
- 인덱스를 지정하지 않은 경우 첫 번째 값만 출력되며, 배열에 저장되어 있는 값을 모두 출력하거나 지정하여 출력할 수 있다.

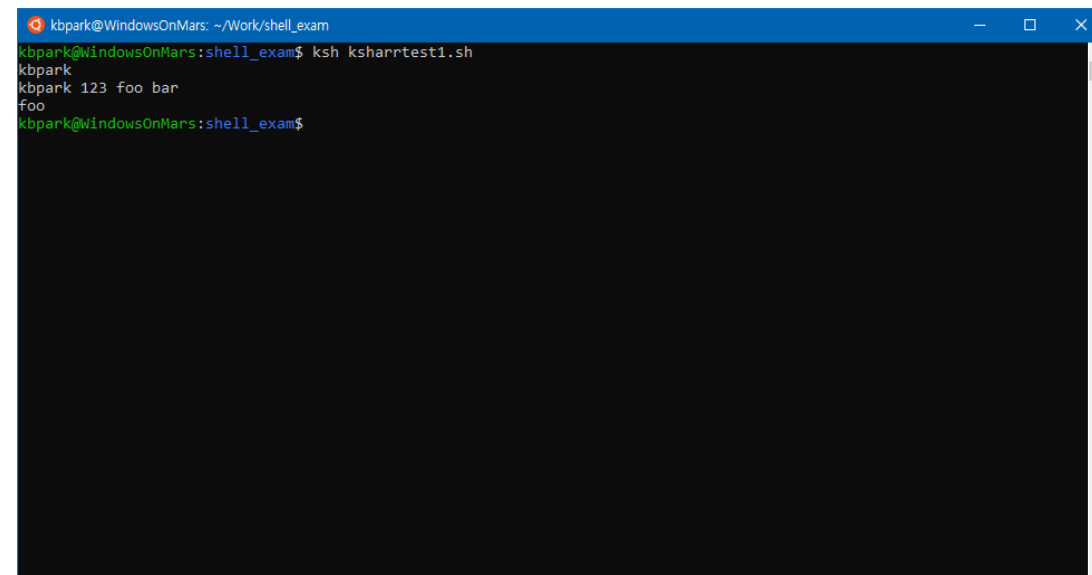
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of a shell script. The prompt is 'kbpark@WindowsOnMars:shell\_exam\$'. The command entered is 'bash ./arrtest1.sh'. The output is: '111', '111 " kbpark "KGA 222 333', and '222'. The prompt returns to 'kbpark@WindowsOnMars:shell\_exam\$'.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./arrtest1.sh
111
111 " kbpark "KGA 222 333
222
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT - 배열

## ksh 배열 정의

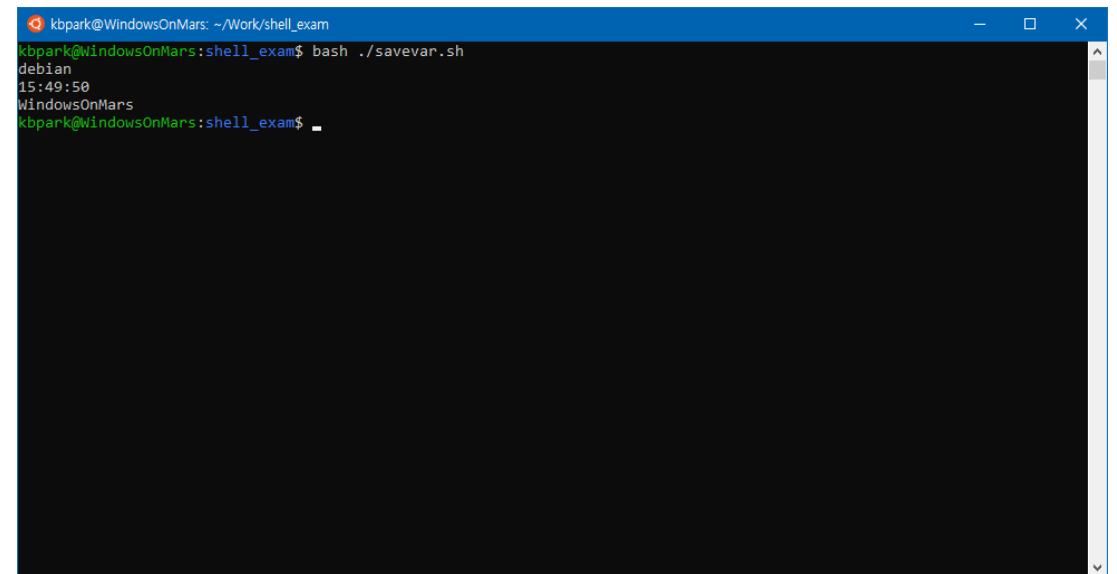
- ksh의 경우 bash와 다르게 set 명령을 사용하여 배열을 지정한다.
- 다음 예제를 확인한다.
  - `#!/bin/ksh`
  - `set -A array "lucas" 123 "foo" "bar"`
  - `echo $array`
  - `echo ${array[*]}`
  - `echo ${array[2]}`

A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of a ksh script. The prompt is 'kbpark@WindowsOnMars:shell\_exam\$'. The user enters 'ksh ksharrtest1.sh'. The script outputs 'kbpark', 'kbpark 123 foo bar', and 'foo' on separate lines. The prompt returns to 'kbpark@WindowsOnMars:shell\_exam\$'.

# LINUX - SHELL SCRIPT - 배열

## 변수 값 혹은 명령 실행 결과 변수 저장하기

- 괄호안에 있는 값을 직접 지정할 수 있지만 변수로 설정한 값을 배열에 저장하거나, 역따옴표(`)를 통해서 명령의 결과를 저장할 수 있다.
  - `#!/bin/bash`
  - `VAR="redhat debian gentoo darwin"`
  - `DISTRO=($VAR)`
  - `echo ${DISTRO[1]}`
  - `TODAY=$(date)`
  - `echo ${TODAY[3]}`
  - `INFO=$(uname -a)`
  - `echo ${INFO[1]}`

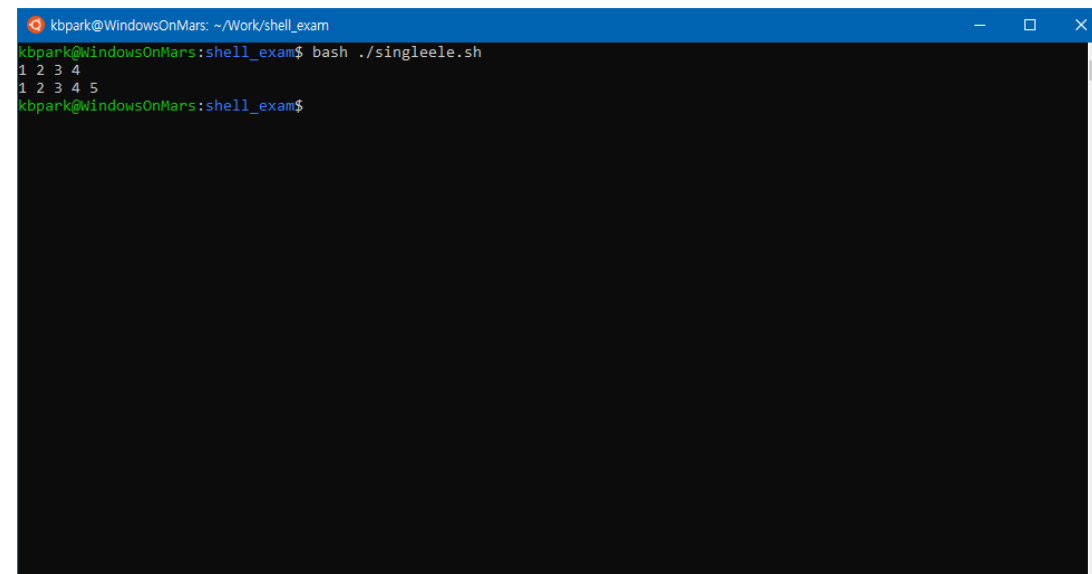


```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./savevar.sh
debian
15:49:50
WindowsOnMars
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT - 배열

## 배열 값 추가

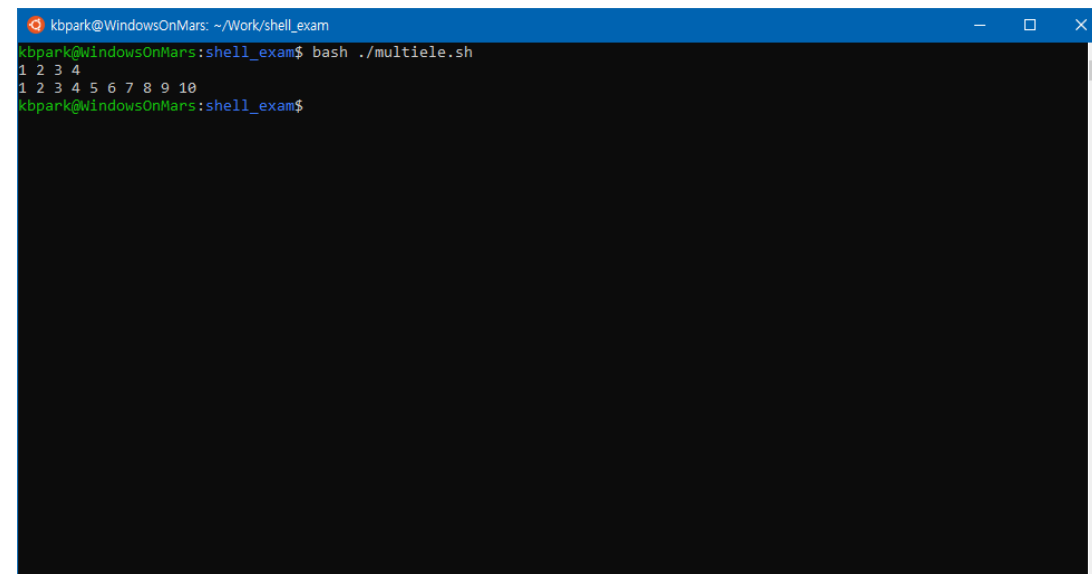
- 먼저 단일 요소를 추가하는 방법을 알아본다.
  - `#!/bin/bash`
  - `NUMBER=(1 2 3 4)`
  - `echo ${NUMBER[*]}`
  - `NUMBER+=(5)`
  - `echo ${NUMBER[*]}`

A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of a shell script. The prompt is 'kbpark@WindowsOnMars:shell\_exam\$'. The command 'bash ./singleleele.sh' is entered. The output shows '1 2 3 4' on the first line and '1 2 3 4 5' on the second line. The prompt returns to 'kbpark@WindowsOnMars:shell\_exam\$'.

# LINUX - SHELL SCRIPT - 배열

## 배열 값 추가

- 여러 요소를 추가하고 싶다면 다음과 같이 수행한다.
  - `#!/bin/bash`
  - `NUMBER=(1 2 3 4)`
  - `echo ${NUMBER[*]}`
  - `NUMBER+=(5 6 7 8 9 10)`
  - `echo ${NUMBER[*]}`

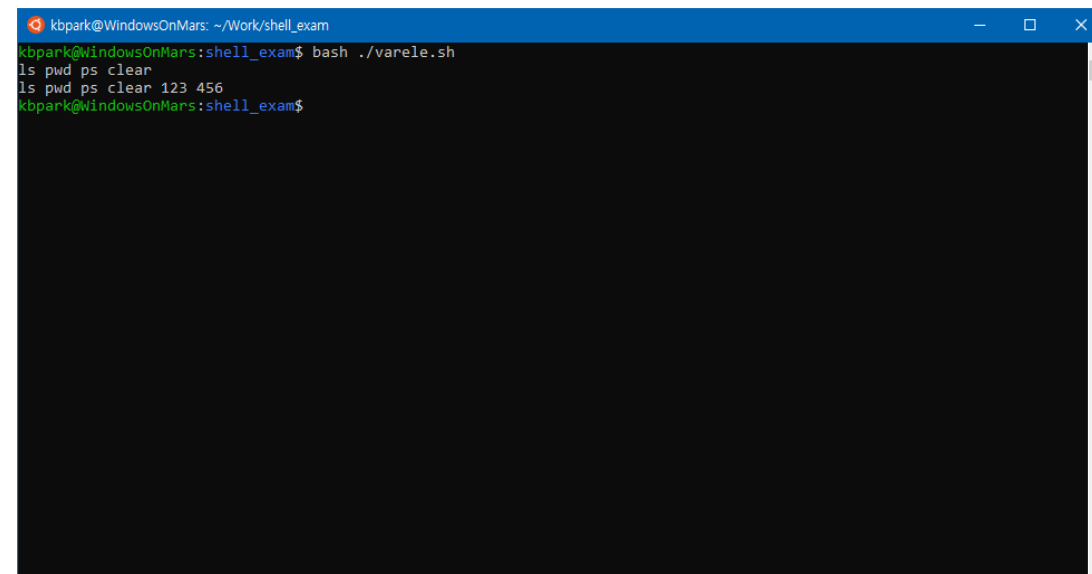
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of a shell script. The prompt is 'kbpark@WindowsOnMars:shell\_exam\$'. The command 'bash ./multiele.sh' is entered. The output shows two lines: '1 2 3 4' and '1 2 3 4 5 6 7 8 9 10'. The prompt returns to 'kbpark@WindowsOnMars:shell\_exam\$'.



# LINUX - SHELL SCRIPT - 배열

## 배열 값 추가

- 변수의 값을 요소로 추가하고자 하면 다음과 같이 수행한다.
  - `#!/bin/bash`
  - `COMMAND=("ls" "pwd" "ps" "clear")`
  - `echo ${COMMAND[*]}`
  - `ELEMENT="123 456"`
  - `COMMAND+=($ELEMENT)`
  - `echo ${COMMAND[*]}`

A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' with a blue header bar. The terminal shows the execution of a shell script. The prompt is 'kbpark@WindowsOnMars:shell\_exam\$'. The first command is 'bash ./varele.sh', which executes three lines: 'ls pwd ps clear', 'ls pwd ps clear 123 456', and then returns to the prompt 'kbpark@WindowsOnMars:shell\_exam\$'. The background of the terminal is black with green text.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./varele.sh
ls pwd ps clear
ls pwd ps clear 123 456
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## 기본 연산자



```
#!/bin/bash

greet() {
    echo "Hello, ${1}"
}

read -p "What is your name: " name
greet "$name"
```

# LINUX - SHELL SCRIPT

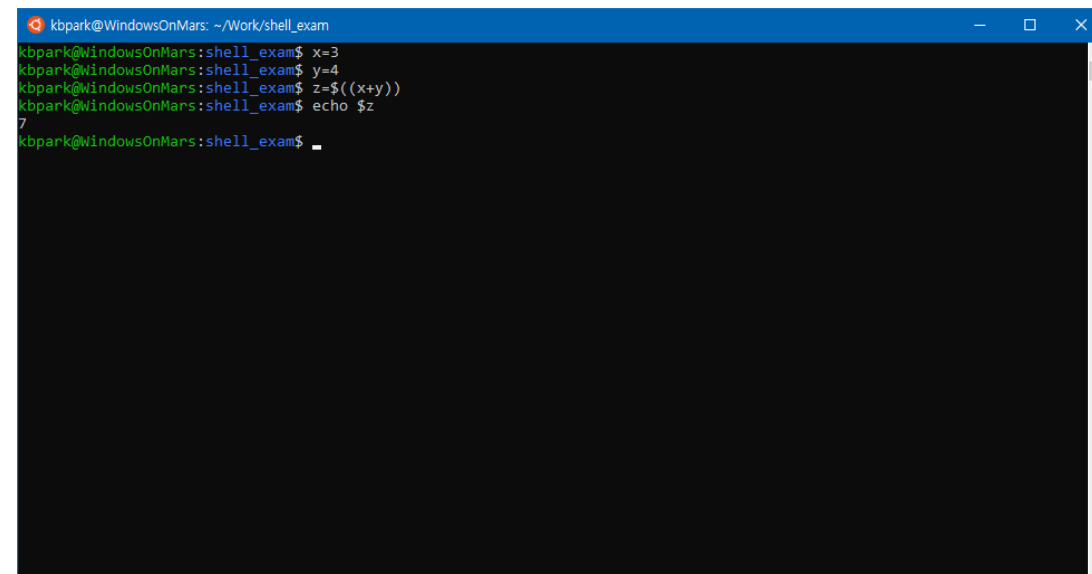
## 기본 연산자

- 다양한 셸 종류에 따라 연산자도 다양하지만 이번 시간에서는 가장 기본적인 bash 셸의 기본 연산자에 대해 설명한다.
  - 산술 연산자
  - 관계 연산자
  - Boolean 연산자
  - 문자열 연산자
  - 파일 테스트 연산자
- bash의 경우 간단한 산술 연산을 수행하는 메커니즘이 존재하지 않기 때문에 awk 혹은 expr과 같은 기본 명령을 사용한다.

# LINUX - SHELL SCRIPT

## bash 변수 처리

- `$ x=3`
- `$ y=4`
- `$ z=$((x+y))`
- `$ echo $z`
- `7`

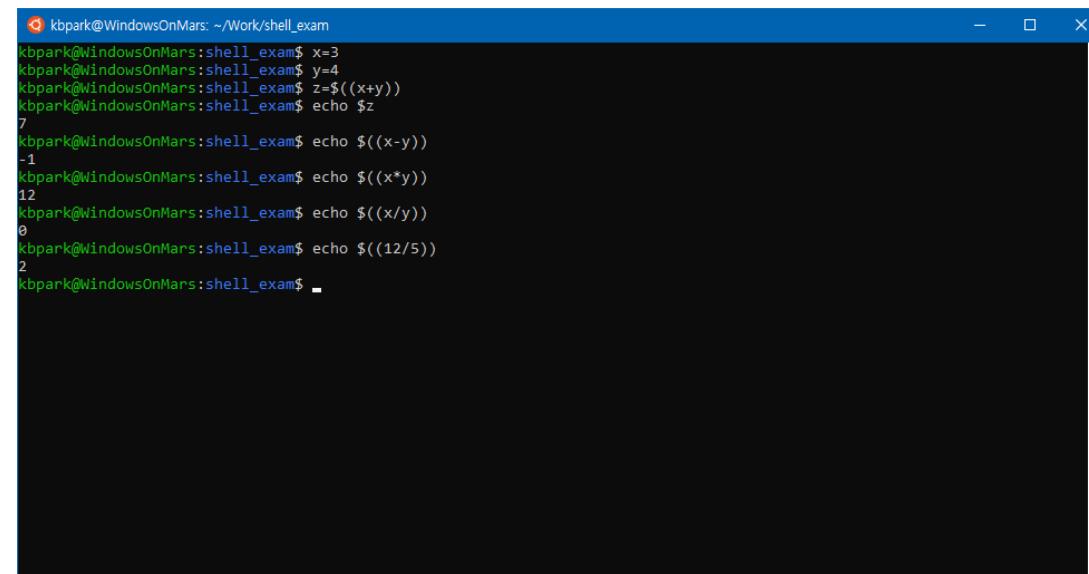
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' with a blue header bar. The terminal shows a series of commands and their outputs: 'x=3', 'y=4', 'z=\$((x+y))', and 'echo \$z'. The output of the last command is '7'.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ x=3
kbpark@WindowsOnMars:shell_exam$ y=4
kbpark@WindowsOnMars:shell_exam$ z=$((x+y))
kbpark@WindowsOnMars:shell_exam$ echo $z
7
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## bash 변수 처리

- `$ x=3`
- `$ y=4`
- `$ z=$((x+y))`
- `$ echo $z`
- `7`
- `echo $((x-y))`
- `echo $((x*y))`
- `echo $((x/y))`
- `# -1`
- `# 12`
- `# 0`
- `echo $((12/5))`
- `# 2`

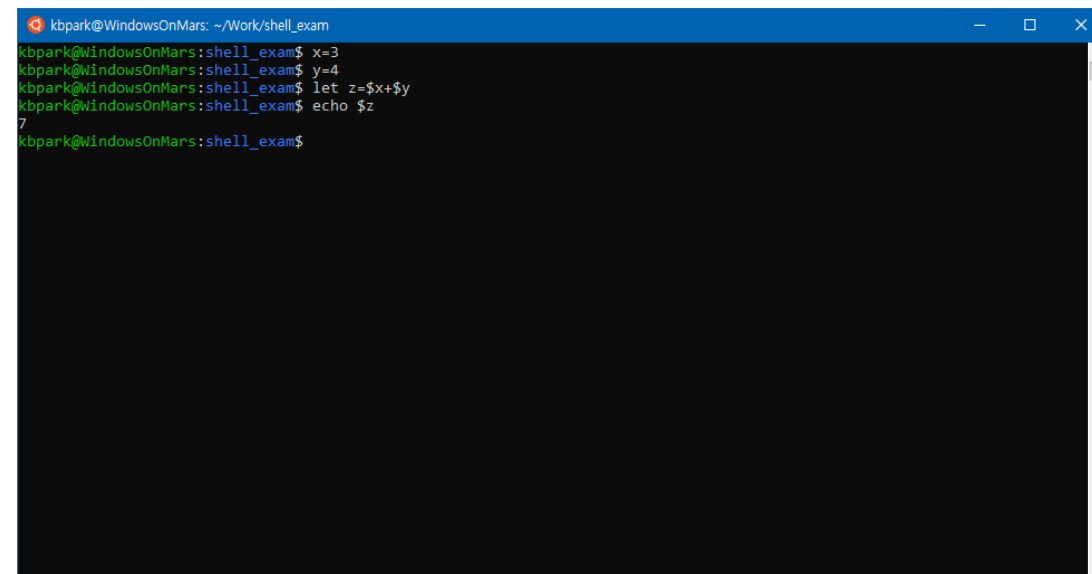
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of a shell script. The commands and their outputs are as follows:

```
kbpark@WindowsOnMars:~/Work/shell_exam$ x=3
kbpark@WindowsOnMars:~/Work/shell_exam$ y=4
kbpark@WindowsOnMars:~/Work/shell_exam$ z=$((x+y))
kbpark@WindowsOnMars:~/Work/shell_exam$ echo $z
7
kbpark@WindowsOnMars:~/Work/shell_exam$ echo $((x-y))
-1
kbpark@WindowsOnMars:~/Work/shell_exam$ echo $((x*y))
12
kbpark@WindowsOnMars:~/Work/shell_exam$ echo $((x/y))
0
kbpark@WindowsOnMars:~/Work/shell_exam$ echo $((12/5))
2
kbpark@WindowsOnMars:~/Work/shell_exam$
```

# LINUX - SHELL SCRIPT

let 명령어

- `$ x=3`
- `$ y=4`
- `$ let z=$x+$y`
- `$ echo $z`
- 7

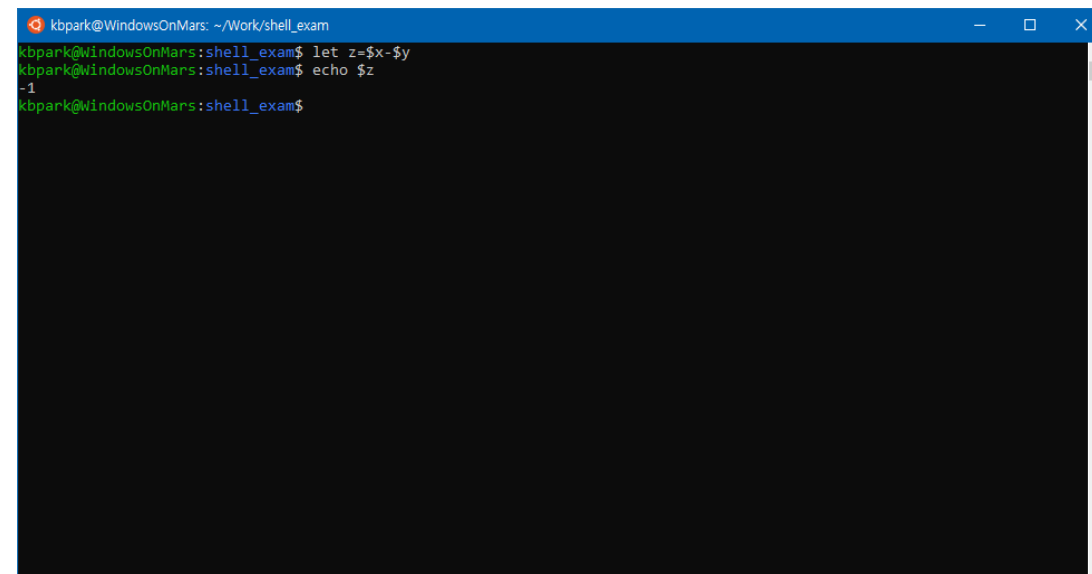
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' with a blue header bar. The terminal shows a sequence of commands and their outputs: 'x=3' is assigned, 'y=4' is assigned, 'let z=\$x+\$y' calculates the sum, 'echo \$z' prints the result '7', and the prompt returns after the final command.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ x=3
kbpark@WindowsOnMars:shell_exam$ y=4
kbpark@WindowsOnMars:shell_exam$ let z=$x+$y
kbpark@WindowsOnMars:shell_exam$ echo $z
7
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

let 명령어

- `let z=$x-$y`
- `echo $z`
- `-1`

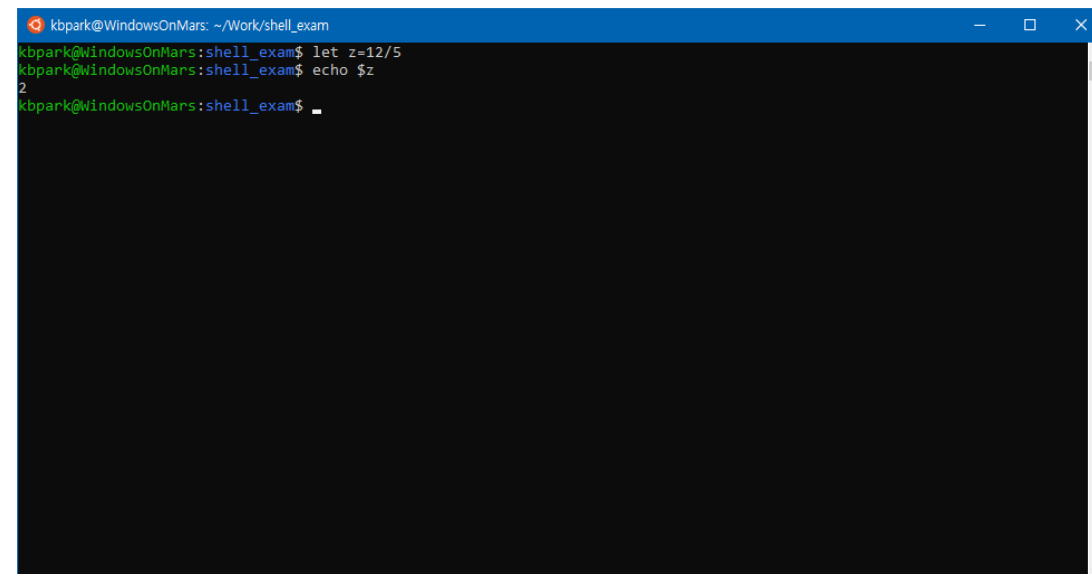
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' with a blue header bar. The terminal shows the execution of a shell script. The first command is 'let z=\$x-\$y', followed by 'echo \$z'. The output of the echo command is '-1'. The prompt 'kbpark@WindowsOnMars:shell\_exam\$' is visible at the end of each line.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ let z=$x-$y
kbpark@WindowsOnMars:shell_exam$ echo $z
-1
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

let 명령어

- let z=12/5
- echo \$z
- # 2

A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' with a blue header bar. The terminal shows the following commands and output:

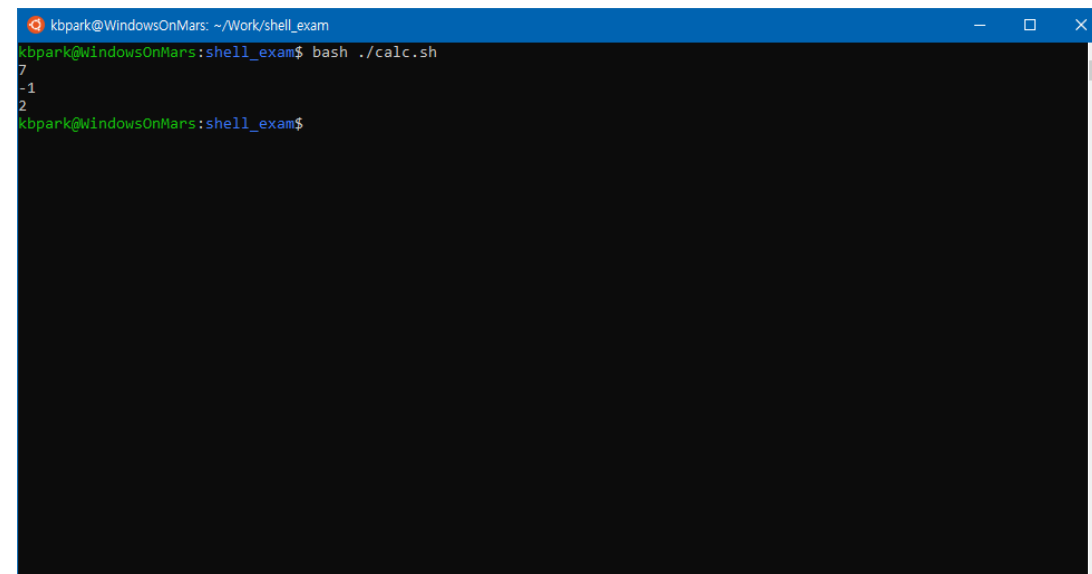
```
kbpark@WindowsOnMars:shell_exam$ let z=12/5
kbpark@WindowsOnMars:shell_exam$ echo $z
2
kbpark@WindowsOnMars:shell_exam$
```



# LINUX - SHELL SCRIPT

## let 명령어

- `#!/bin/bash`
- `x=3`
- `y=4`
- `let z=$x+$y`
- `echo $z`
- `let z=$x-$y`
- `echo $z`
- `let z=12/5`
- `echo $z`

A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' with a blue header bar. The terminal shows a shell script being executed. The prompt is 'kbpark@WindowsOnMars:shell\_exam\$'. The command entered is 'bash ./calc.sh'. The output of the script is displayed on the next line: '7', '-1', and '2'. The prompt returns to 'kbpark@WindowsOnMars:shell\_exam\$'.

# LINUX - SHELL SCRIPT

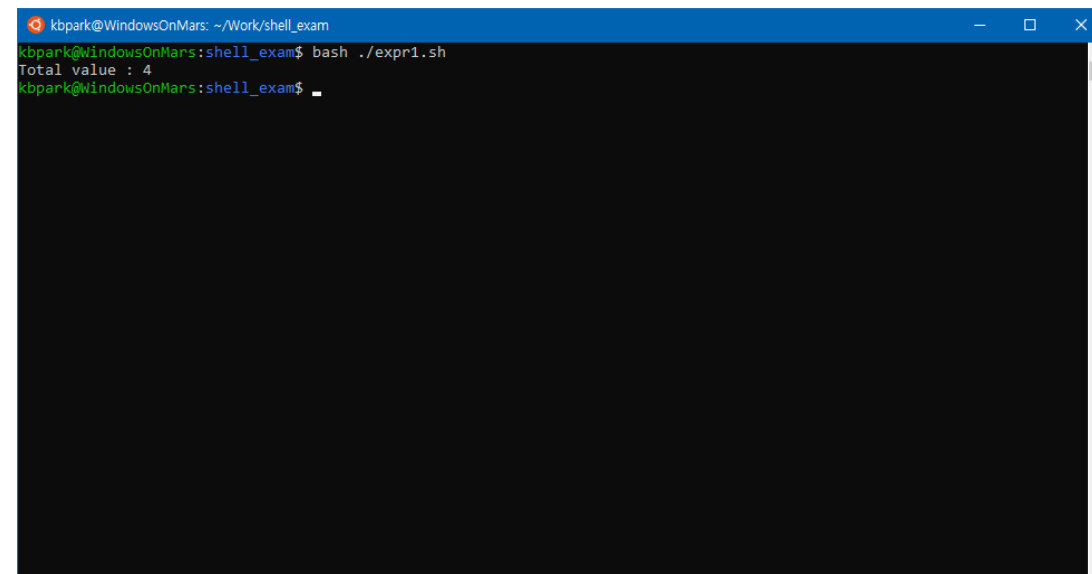
expr 명령어

- 숫자와 연산자 사이를 반드시 띄어 써야 한다.
- 예를 들어  $2+2$  형식으로 하면 동작하지 않으며  $2 + 2$  형식으로 작성해야 한다.
- 또한 쉘 스크립트 내에서 연산을 적용하기 위해서는 `` backtick을 추가해야 한다.

# LINUX - SHELL SCRIPT

expr 명령어

- `#!/bin/sh`
- `val=`expr 2 + 2``
- `echo "Total value : $val"`

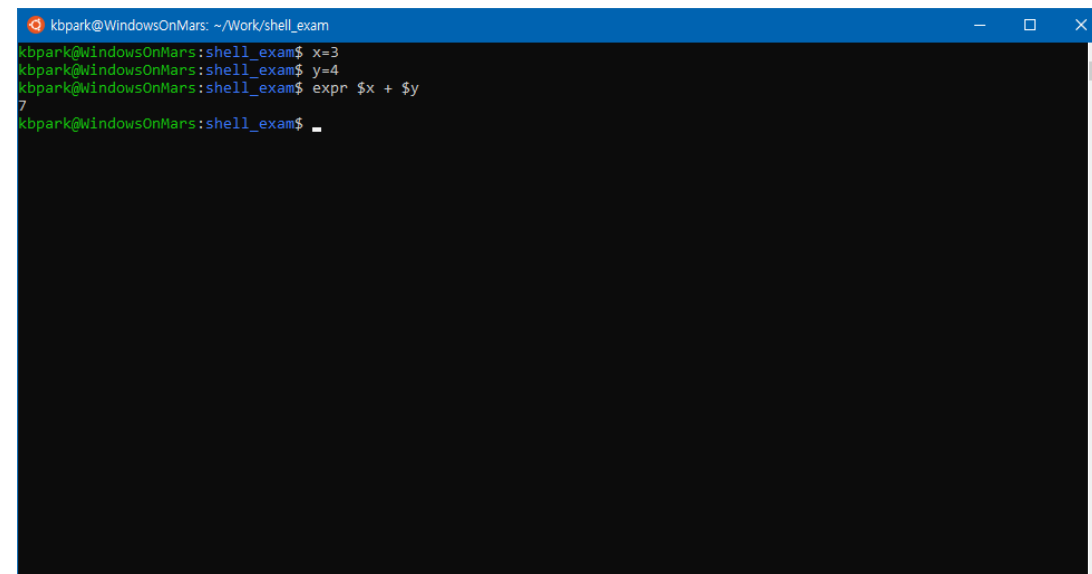
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' with a blue header bar. The terminal shows the execution of a shell script. The prompt is 'kbpark@WindowsOnMars:shell\_exam\$'. The user enters 'bash ./expr1.sh'. The script outputs 'Total value : 4'. The prompt returns to 'kbpark@WindowsOnMars:shell\_exam\$'.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./expr1.sh
Total value : 4
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

expr 명령어

- `x=3`
- `y=4`
- `expr $x + $y`

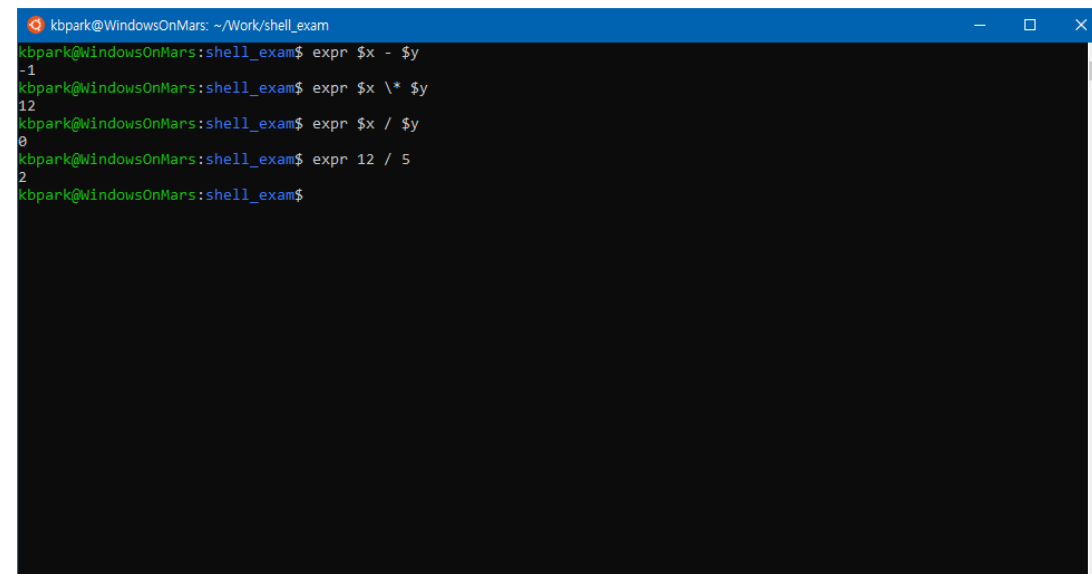
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' with a blue header bar. The terminal shows a sequence of commands and their outputs: 'x=3' is entered and executed; 'y=4' is entered and executed; 'expr \$x + \$y' is entered and executed, resulting in the output '7'. The prompt 'kbpark@WindowsOnMars:shell\_exam\$' is visible at the end of each line.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ x=3
kbpark@WindowsOnMars:shell_exam$ y=4
kbpark@WindowsOnMars:shell_exam$ expr $x + $y
7
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## expr 명령어

- `expr $x - $y`
- `expr $x % $y`
- `expr $x / $y`
- `# -1`
- `# 12`
- `# 0`
- `expr 12 / 5`
- `# 2`

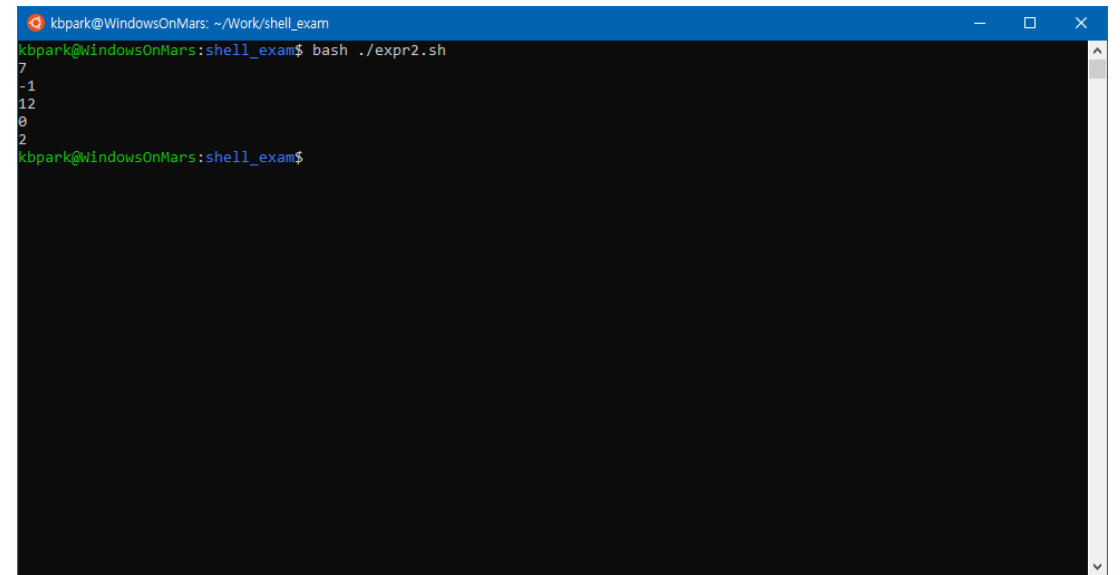
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of the 'expr' command. The commands and their outputs are: 'expr \$x - \$y' returns '-1', 'expr \$x % \$y' returns '12', 'expr \$x / \$y' returns '0', and 'expr 12 / 5' returns '2'.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ expr $x - $y
-1
kbpark@WindowsOnMars:shell_exam$ expr $x % $y
12
kbpark@WindowsOnMars:shell_exam$ expr $x / $y
0
kbpark@WindowsOnMars:shell_exam$ expr 12 / 5
2
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## expr 명령어

- `expr $x - $y`
- `expr $x % $y`
- `expr $x / $y`
- `# -1`
- `# 12`
- `# 0`
- `expr 12 / 5`
- `# 2`

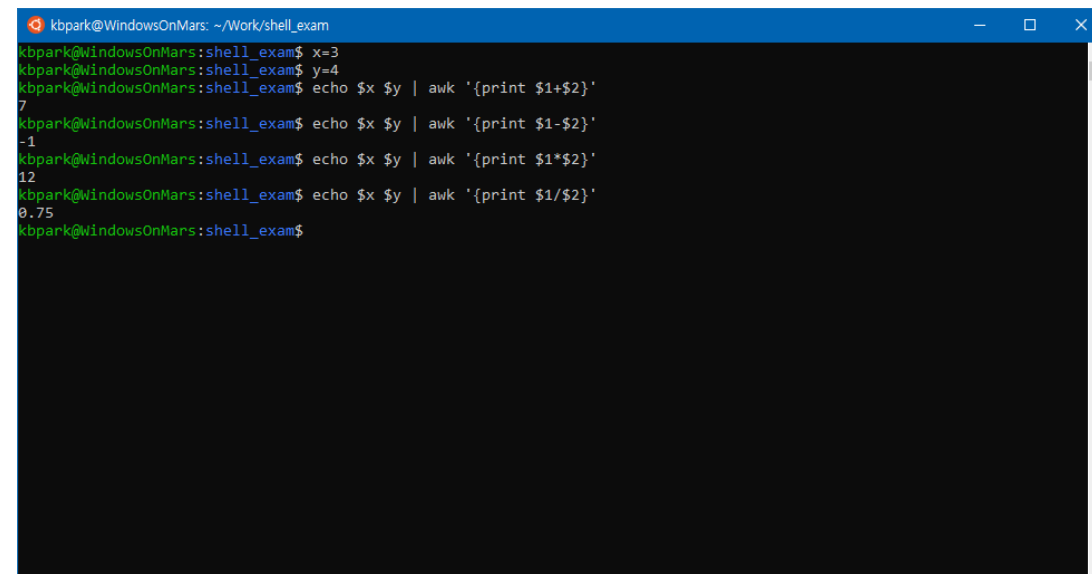


```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./expr2.sh
7
-1
12
0
2
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## awk 명령어

- 리눅스에서 기본적으로 제공하는 명령어로 소수점 계산이 가능하다.
- 단 직접 변수를 사용하지 못하기 때문에 파이프를 통해 전달하는 과정이 필요하다.
  - `x=3`
  - `y=4`
  - `echo $x $y | awk '{print $1+$2}'`
  - `# 7`
  - `echo $x $y | awk '{print $1-$2}'`
  - `# -1`
  - `echo $x $y | awk '{print $1*$2}'`
  - `# 12`
  - `echo $x $y | awk '{print $1/$2}'`
  - `# 0.75`



```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ x=3
kbpark@WindowsOnMars:shell_exam$ y=4
kbpark@WindowsOnMars:shell_exam$ echo $x $y | awk '{print $1+$2}'
7
kbpark@WindowsOnMars:shell_exam$ echo $x $y | awk '{print $1-$2}'
-1
kbpark@WindowsOnMars:shell_exam$ echo $x $y | awk '{print $1*$2}'
12
kbpark@WindowsOnMars:shell_exam$ echo $x $y | awk '{print $1/$2}'
0.75
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## bc 명령어

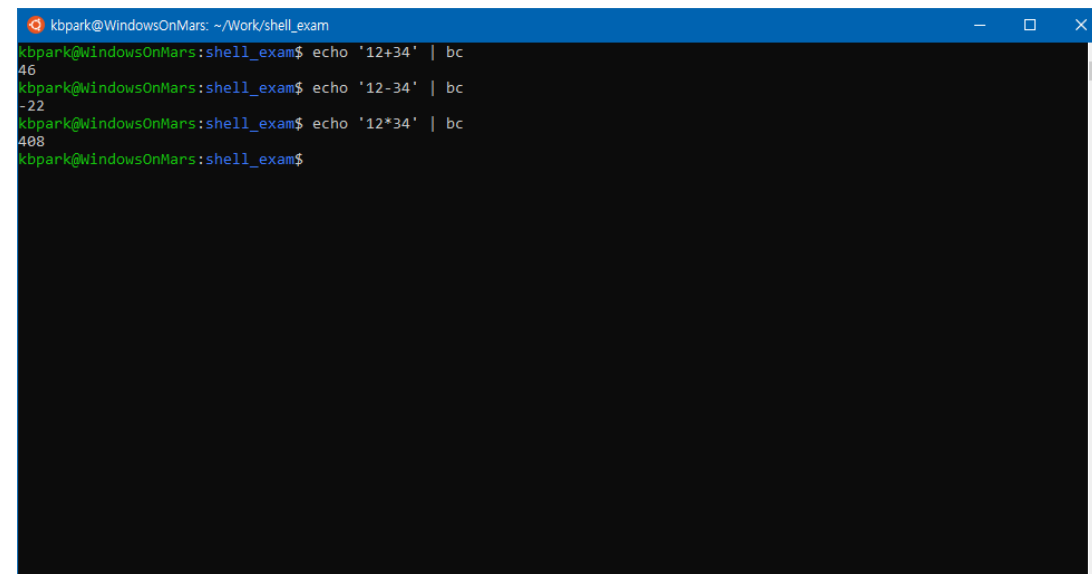
- basic calculator의 약자로 리눅스 bc가 설치되어야 한다.
- 인터랙티브 모드와 배치 모드 둘다 사용이 가능하고, 실수, 사칙연산, 거듭제곱 등의 연산과 같은 고급 기능이 있으며 가법다는 특징이 있다.



# LINUX - SHELL SCRIPT

## bc 명령어

- `$ echo '12+34' | bc`
- 46
- `$ echo '12-34' | bc`
- -22
- `$ echo '12*34' | bc`
- 408

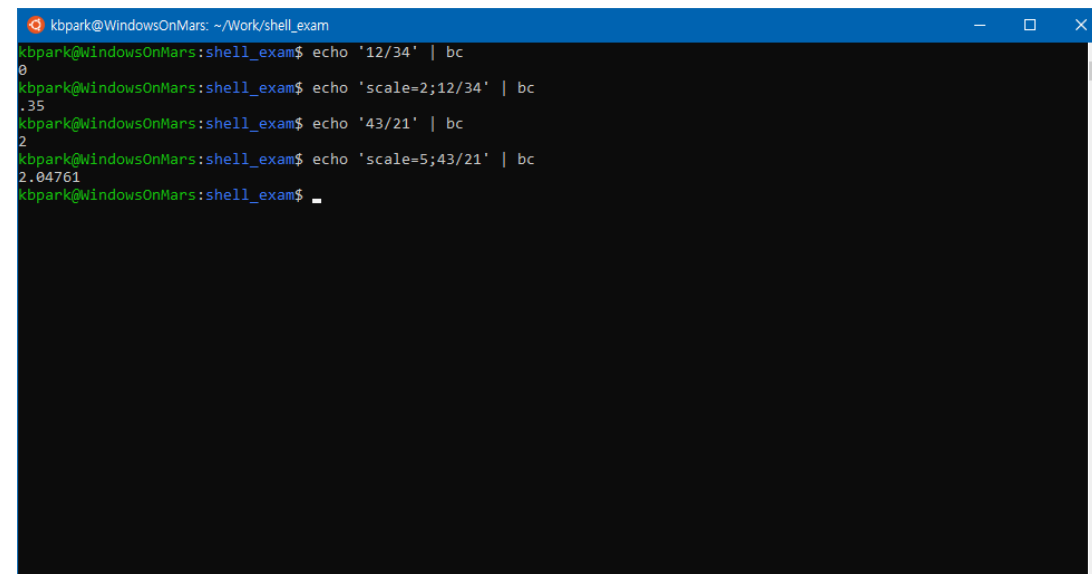
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing three lines of commands and their outputs. The first line is 'kbpark@WindowsOnMars:shell\_exam\$ echo '12+34' | bc' with output '46'. The second line is 'kbpark@WindowsOnMars:shell\_exam\$ echo '12-34' | bc' with output '-22'. The third line is 'kbpark@WindowsOnMars:shell\_exam\$ echo '12\*34' | bc' with output '408'. The prompt 'kbpark@WindowsOnMars:shell\_exam\$' is visible at the bottom.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ echo '12+34' | bc
46
kbpark@WindowsOnMars:shell_exam$ echo '12-34' | bc
-22
kbpark@WindowsOnMars:shell_exam$ echo '12*34' | bc
408
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## bc 명령어

- `$ echo '12/34' | bc`
- 0
- `$ echo 'scale=2;12/34' | bc`
- .35
- `$ echo '43/21' | bc`
- 2
- `$ echo 'scale=5;43/21' | bc`
- 2.04761

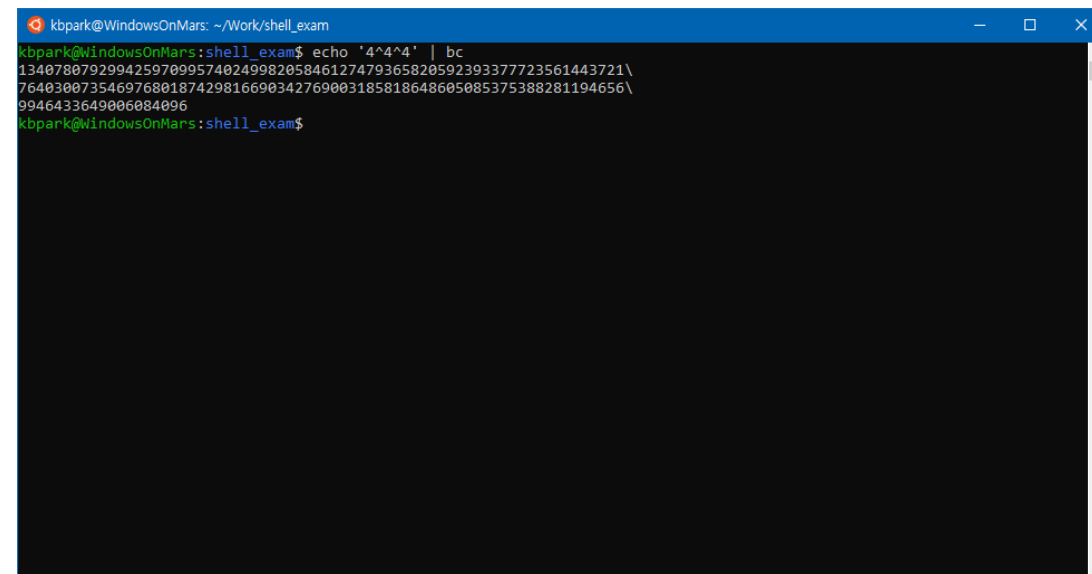
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of the 'bc' command. The window has a blue title bar and a black background with green text. The commands and their outputs are as follows:

```
kbpark@WindowsOnMars:~/Work/shell_exam$ echo '12/34' | bc
0
kbpark@WindowsOnMars:~/Work/shell_exam$ echo 'scale=2;12/34' | bc
.35
kbpark@WindowsOnMars:~/Work/shell_exam$ echo '43/21' | bc
2
kbpark@WindowsOnMars:~/Work/shell_exam$ echo 'scale=5;43/21' | bc
2.04761
kbpark@WindowsOnMars:~/Work/shell_exam$
```

# LINUX - SHELL SCRIPT

bc 명령어

- `$ echo '4^4^4' | bc`
  - 13407807929942597099574024998205846127479365  
820592393377723561443721W  
76403007354697680187429816690342769003185818648  
605085375388281194656W  
9946433649006084096



```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ echo '4^4^4' | bc
13407807929942597099574024998205846127479365820592393377723561443721\
76403007354697680187429816690342769003185818648605085375388281194656\
9946433649006084096
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## 산술 연산자

Operator	Description	Example
<b>+</b> (Addition)	덧셈	expr \$a + \$b
<b>-</b> (Subtraction)	뺄셈	expr \$a - \$b
<b>*</b> (Multiplication)	곱셈	expr \$a * \$b
<b>/</b> (Division)	나누기	expr \$b / \$a
<b>%</b> (Modulus)	나머지 반환	expr \$b % \$a
<b>=</b> (Assignment)	값 할당	a = \$b
<b>==</b> (Equality)	비교	[ \$a == \$b ]
<b>!=</b> (Not Equality)	비교	[ \$a != \$b ]

## 관계 연산자

Operator	Description	Example
<b>-eq</b>	두 피연산자의 값이 동일한 여부를 확인하여 같으면 참을 반환한다.	[ \$a -eq \$b ]
<b>-ne</b>	두 피연산자의 값이 동일한지 여부를 확인하여 같지 않으면 참을 반환한다.	[ \$a -ne \$b ]
<b>-gt</b>	왼쪽 피연산자의 값이 오른쪽 피연산자의 값보다 큰지 확인하고, 그렇다면 조건이 참이된다.	[ \$a -gt \$b ]
<b>-lt</b>	왼쪽 피연산자의 값이 오른쪽 피연산자의 값보다 작은지 확인하고, 그렇다면 조건이 참이된다.	[ \$a -lt \$b ]
<b>-ge</b>	왼쪽 피연산자의 값이 오른쪽 피연산자의 값보다 크거나 같은지 확인하고, 그렇다면 조건이 참이된다.	[ \$a -ge \$b ]
<b>-le</b>	왼쪽 피연산자의 값이 오른쪽 피연산자의 값보다 작거나 같은지 확인하고, 그렇다면 조건이 참이된다.	[ \$a -le \$b ]

bash는 숫자 값과 관련된 관계 연산을 제공하며, 문자열은 제공되지 않는다.

# LINUX - SHELL SCRIPT

## Boolean 연산자

Operator	Description	Example
!	논리 부정	[ ! false ] is true.
-o	OR 연산자	[ \$a -lt 20 -o \$b -gt 100 ] is true.
-a	AND 연산자	[ \$a -lt 20 -a \$b -gt 100 ] is false.

a가 10이고 b가 20으로 가정한다.

## 문자열 연산자

Operator	Description	Example
=	두 피연산자의 값을 확인하고 같다면 참이다.	[ \$a = \$b ]
!=	두 피연산자의 값을 확인하고 같지 않으면 참이다.	[ \$a != \$b ]
-z	주어진 문자열 피연산자의 길이가 0이면 참이다.	[ -z \$a ]
-n	주어진 문자열 피연산자의 0이 아니면 참이다.	[ -n \$a ]
str	빈 문자열인지를 확인하고 비어있으면 거짓을 반환한다.	[ \$a ]

bash에서 문자열 연산을 위해서는 다음과 같은 명령을 수행해야 한다.  
변수 a에는 "abc"가 있고 변수 b에는 "efg"가 있다고 가정한다.

# LINUX - SHELL SCRIPT

## 파일 테스트 연산자

- 파일과 관련된 속성을 테스트 하는데 확인하는 연산자이다.
- 파일의 이름은 test이고 크기가 100Byte, R/W/X 권한이 있다고 가정한다.

Operator	Description	Example
-b file	파일이 블록 파일인지 확인한다. (블록 디바이스 등)	[ -b \$file ]
-c file	파일이 문자 파일인지 확인한다. (키보드, 모뎀, 사운드 카드 등)	[ -c \$file ]
-d file	파일이 디렉터리인지 확인한다.	[ -d \$file ]
-f file	파일이 디렉터리나 일반파일인지 확인한다. (장치 파일이 아님)	[ -f \$file ]
-g file	파일에 SGID (Set Group ID)가 설정되어 있는지 확인한다.	[ -g \$file ]
-k file	파일에 고정 비트가 설정되어 있는지 확인한다.	[ -k \$file ]
-p file	파일이 명명 파이프인지 확인한다.	[ -p \$file ]
-t file	파일 설명자가 있고 터미널과 연결되어 있는지 확인한다.	[ -t \$file ]
-u file	파일에 SUID (Set User ID) 비트가 설정되어 있는지 확인한다.	[ -u \$file ]
-r file	파일을 읽을 수 있는지 확인한다.	[ -r \$file ]
-w file	파일 쓰기가 가능한지 확인한다.	[ -w \$file ]
-x file	파일 실행이 가능한지 확인한다.	[ -x \$file ]
-s file	파일의 크기가 0보다 큰지 확인한다.	[ -s \$file ]
-e file	파일이 존재하는지 확인한다.	[ -e \$file ]

# LINUX - SHELL SCRIPT

## 연산자 - Shell Decision Making

- 이번에는 Unix Shell에서 특정 조건일 때, 올바른 수행이 가능하도록 하는 조건문에 대해서 알아본다.

## The if...else statements

- if else 문은 주어진 옵션 집합에서 조건을 선택할 수 있도록 지원한다.
- 어떠한 조건에 대해서 True가 될 때 지정된 문이 실행되고, False일 경우 실행되지 않는다.
- 대부분 비교 연산자를 통해 작성한다.
- 각 구문에 대한 공백을 지켜야 오류가 발생하지 않는다.

# LINUX - SHELL SCRIPT

if...fi statement

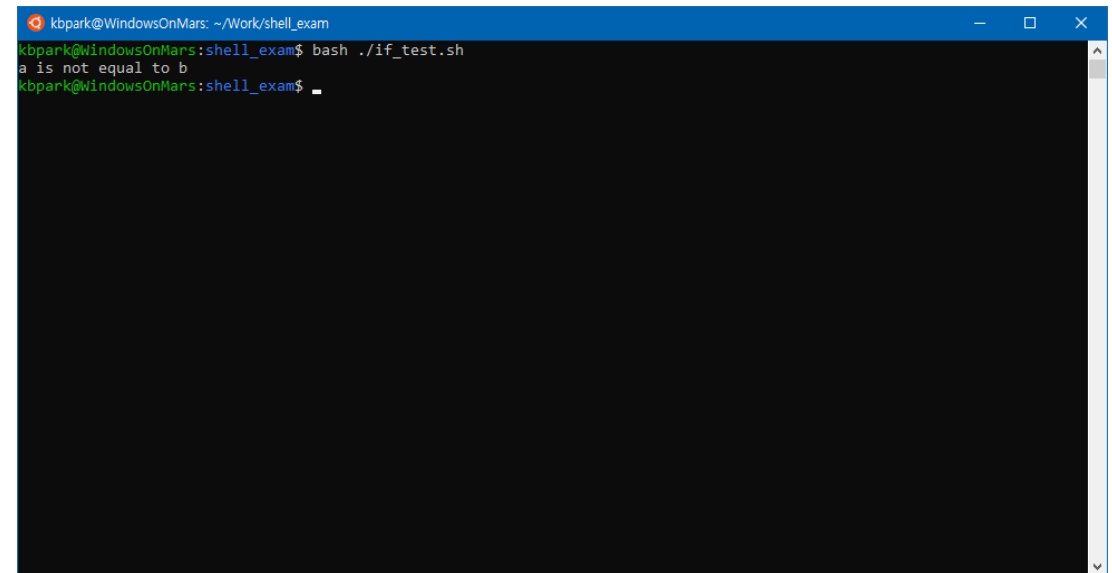
- 문법
  - if [ expression ]
  - then
  - Statement(s) to be executed if expression is true
  - fi



# LINUX - SHELL SCRIPT

## if...fi statement

- `#!/bin/bash`
- `a=10`
- `b=20`
- `if [ $a == $b ]`
- `then`
- `echo "a is equal to b"`
- `fi`
- `if [ $a != $b ]`
- `then`
- `echo "a is not equal to b"`
- `fi`

A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' with a blue header bar. The terminal shows the command 'bash ./if\_test.sh' being executed. The output is 'a is not equal to b' on a new line. The prompt returns to 'kbpark@WindowsOnMars:shell\_exam\$' with a cursor. The background of the terminal is black with green text.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./if_test.sh
a is not equal to b
kbpark@WindowsOnMars:shell_exam$ _
```

# LINUX - SHELL SCRIPT

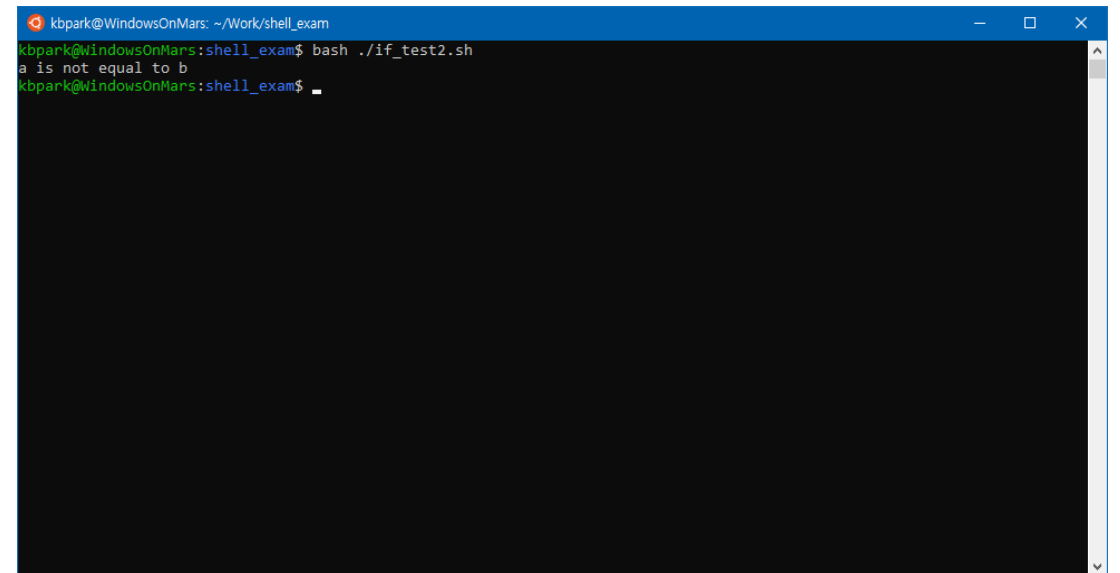
if...else...fi statement

- if [ expression ]
- then
- Statement(s) to be executed if expression is true
- else
- Statement(s) to be executed if expression is not true
- fi

# LINUX - SHELL SCRIPT

## if...else...fi statement

- `#!/bin/bash`
- `a=10`
- `b=20`
- `if [ $a == $b ]`
- `then`
- `echo "a is equal to b"`
- `else`
- `echo "a is not equal to b"`
- `fi`

A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' with a blue header bar. The terminal shows the command 'bash ./if\_test2.sh' being executed. The output is 'a is not equal to b'. The prompt returns to 'kbpark@WindowsOnMars:shell\_exam\$' with a cursor.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./if_test2.sh
a is not equal to b
kbpark@WindowsOnMars:shell_exam$ _
```

# LINUX - SHELL SCRIPT

## if...elif...fi statement

- if [ expression 1 ]
- then
- Statement(s) to be executed if expression 1 is true
- elif [ expression 2 ]
- then
- Statement(s) to be executed if expression 2 is true
- elif [ expression 3 ]
- then
- Statement(s) to be executed if expression 3 is true
- else
- Statement(s) to be executed if no expression is true
- fi

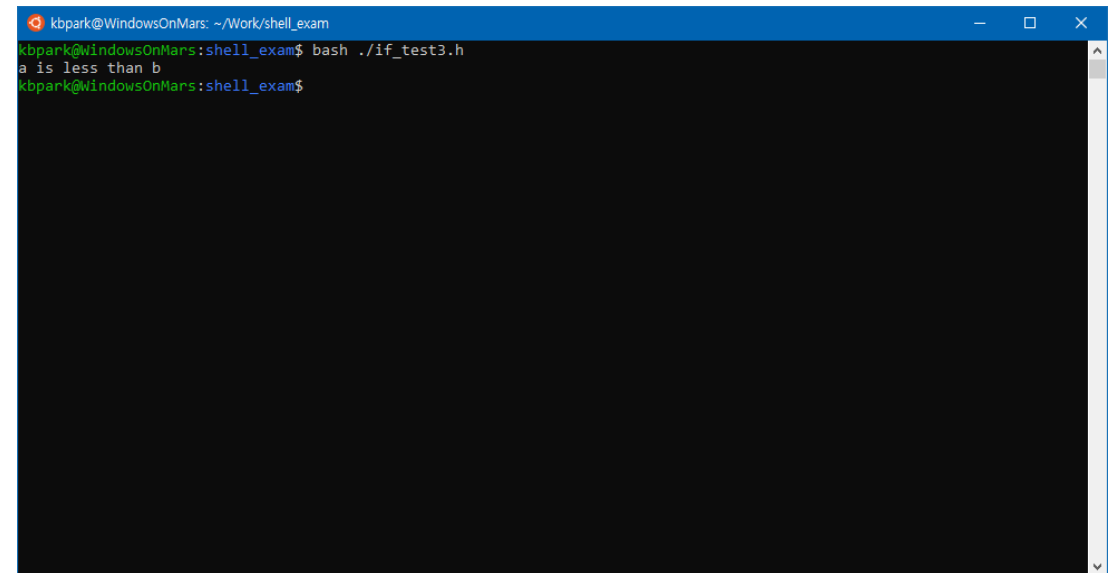
# LINUX - SHELL SCRIPT

## if...elif...fi statement

```
#!/bin/bash

a=10
b=20

if [ $a == $b ]
then
    echo "a is equal to b"
elif [ $a -gt $b ]
then
    echo "a is greater than b"
elif [ $a -lt $b ]
then
    echo "a is less than b"
else
    echo "None of the condition met"
fi
```

A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of a shell script. The prompt is 'kbpark@WindowsOnMars:shell\_exam\$'. The user enters 'bash ./if\_test3.h'. The output is 'a is less than b'. The prompt returns to 'kbpark@WindowsOnMars:shell\_exam\$'.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./if_test3.h
a is less than b
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## The case...esac Statement

```
case word in
  pattern1)
    Statement(s) to be executed if pattern1 matches
    ;;
  pattern2)
    Statement(s) to be executed if pattern2 matches
    ;;
  pattern3)
    Statement(s) to be executed if pattern3 matches
    ;;
  *)
    Default condition to be executed
    ;;
esac
```

- 문자열 word는 일치하는 항목이 발견될 때까지 모든 패턴과 비교를 수행하며, 일치한 패턴이 있을 경우 해당 작업을 수행한다. 만약 일치하는 항목이 없으면 아무 작업을 수행하지 않고 종료된다.
- 최대의 패턴의 수는 정해져 있지 않지만 최소 하나 이상의 패턴이 존재해야 하며, ;;의 경우 C/C++ 프로그래밍에서 break와 동일한 기능을 수행한다.

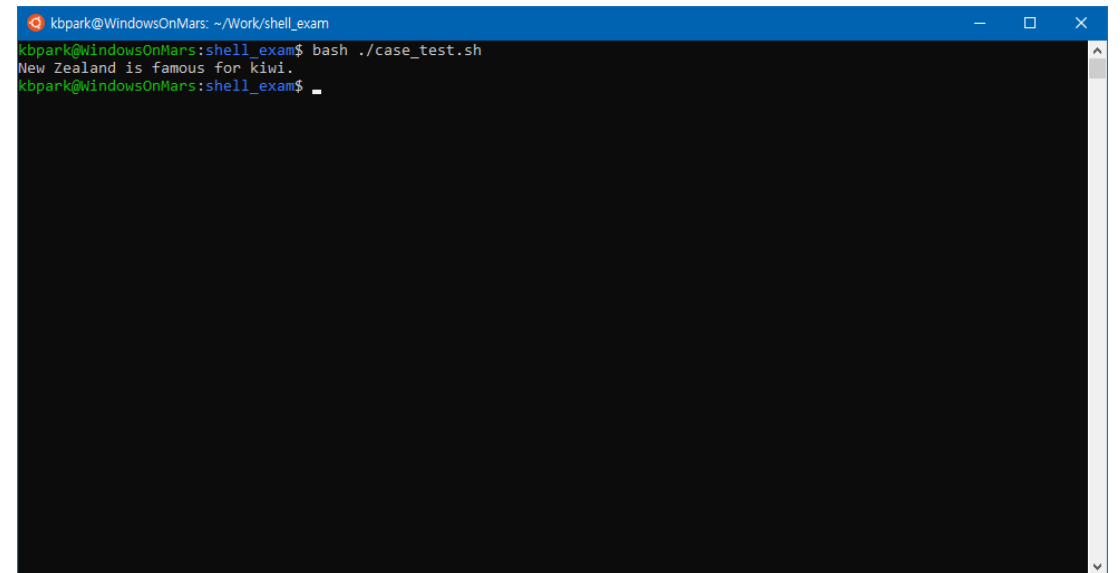
# LINUX - SHELL SCRIPT

## The case...esac Statement

```
#!/bin/bash

FRUIT="kiwi"

case "$FRUIT" in
    "apple") echo "Apple pie is quite tasty."
    ;;
    "banana") echo "I like banana nut bread."
    ;;
    "kiwi") echo "New Zealand is famous for kiwi."
    ;;
esac
```

A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of a shell script. The prompt is 'kbpark@WindowsOnMars:shell\_exam\$'. The user enters 'bash ./case\_test.sh'. The script outputs 'New Zealand is famous for kiwi.' and the prompt returns to 'kbpark@WindowsOnMars:shell\_exam\$' with a cursor.

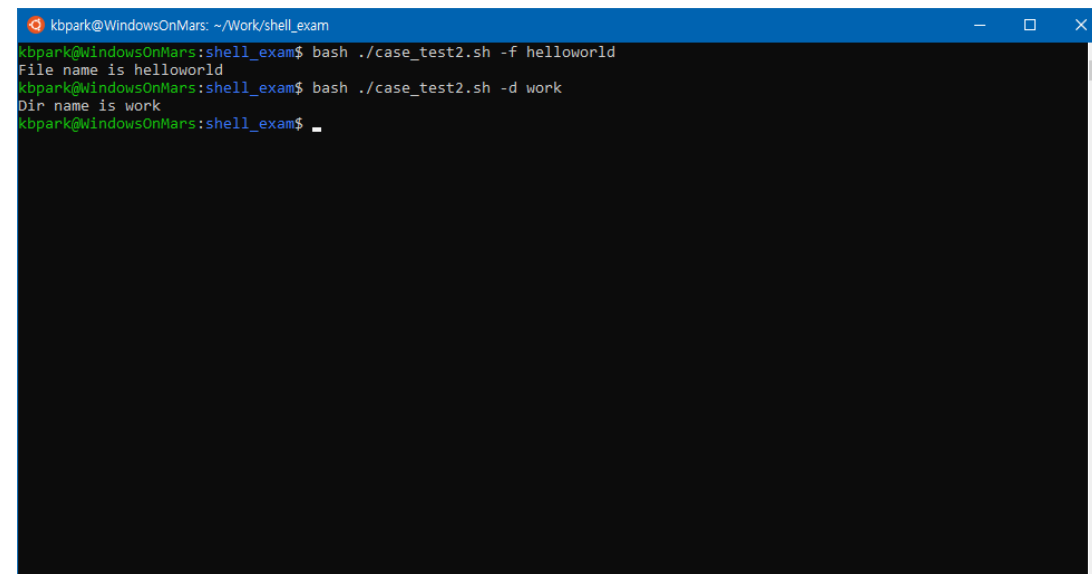
```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./case_test.sh
New Zealand is famous for kiwi.
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## The case...esac Statement

```
#!/bin/bash

option="${1}"
case ${option} in
    -f) FILE="${2}"
        echo "File name is $FILE"
        ;;
    -d) DIR="${2}"
        echo "Dir name is $DIR"
        ;;
    *)
        echo "`basename ${0}`:usage: [-f file] | [-d directory]"
        exit 1 # Command to come out of the program with status 1
        ;;
esac
```

A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of a shell script. The user runs 'bash ./case\_test2.sh -f helloworld', and the script outputs 'File name is helloworld'. Then, the user runs 'bash ./case\_test2.sh -d work', and the script outputs 'Dir name is work'. The prompt returns to the user.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./case_test2.sh -f helloworld
File name is helloworld
kbpark@WindowsOnMars:shell_exam$ bash ./case_test2.sh -d work
Dir name is work
kbpark@WindowsOnMars:shell_exam$
```

커멘트 라인에서 인수를 받아서 처리해야 하는 경우 유용하게 사용될 수 있다.



# LINUX - SHELL SCRIPT

## Shell Loop Types

- 이번에는 Unix Shell에서 사용하는 반복문에 대해서 알아본다.
- 반복은 일련의 명령을 반복할 수 있도록 하는 프로그래밍 도구이다.
- 각각의 반복문은 상황에 따라서 적절하게 선택할 수 있어야 한다.

# LINUX - SHELL SCRIPT

## The while loop

- while 문의 기본 문법은 다음과 같다.
  - 조건이 참인 동안 명령1과 명령2가 순차적으로 반복된다.
  - 명령을 처리 하는 중간에 if문과 continue, break 문을 이용하여 while문의 처음으로 돌아가거나, 탈출하는 것이 가능하다.
- while [ 조건 ]
  - do
  - 명령1
  - 명령2
  - done

# LINUX - SHELL SCRIPT

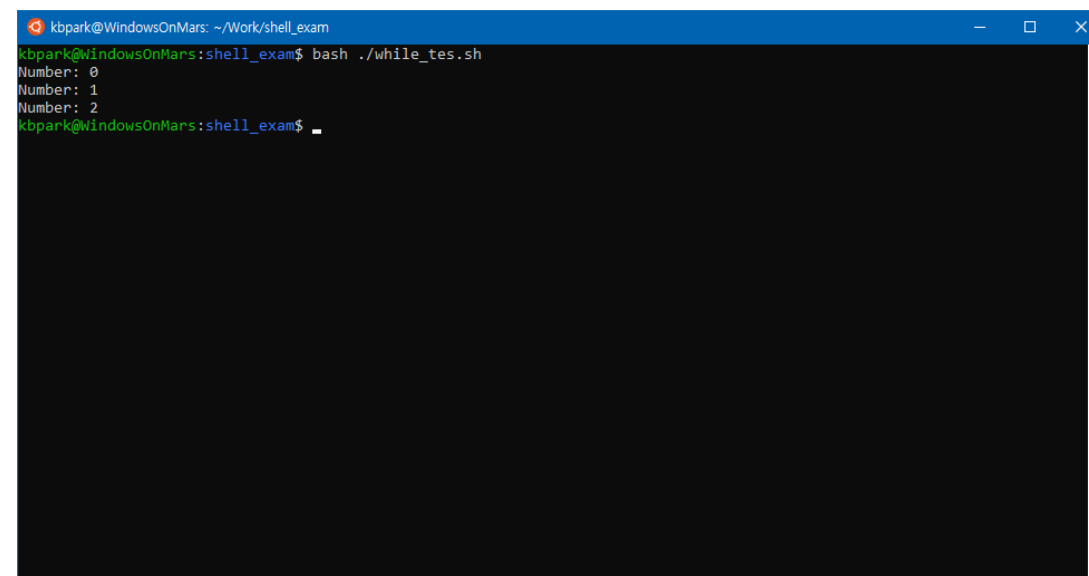
## The while loop

- while문을 한줄로 작성하는 방법은 다음과 같다.
- `while [ 조건 ]; do 명령1;명령2; done`

# LINUX - SHELL SCRIPT

## The while loop

- 기본적인 루프문 처리는 다음과 같다. number가 2보다 작거나 같을 동안(≤) 반복됩니다.
  - `#!/bin/bash`
  - `number=0`
  - `while [ $number -le 2 ]`
  - `do`
  - `echo "Number: ${number}"`
  - `((number++))`
  - `done`

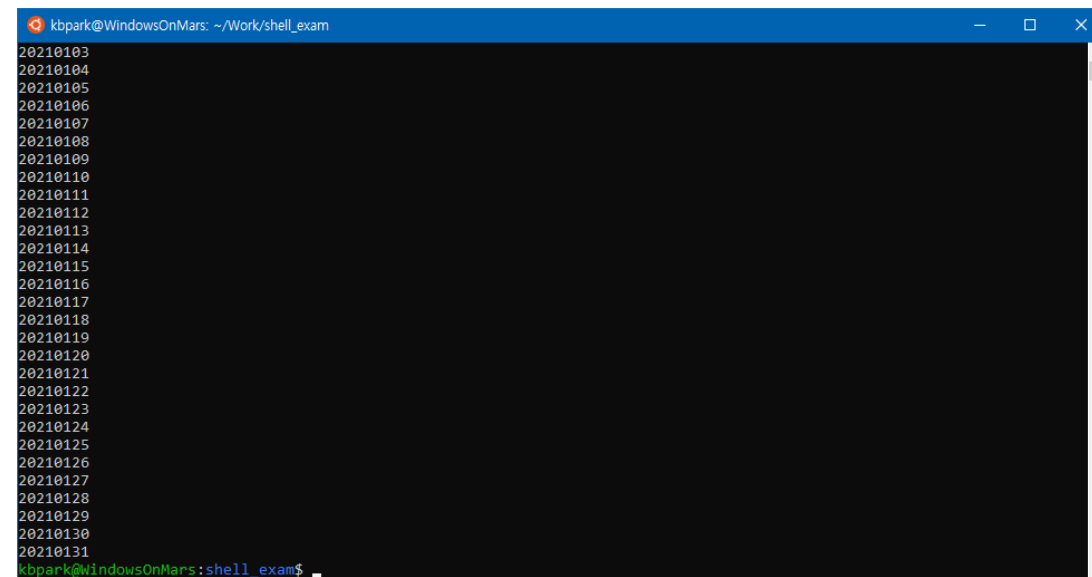
A terminal window titled 'kbpark@WindowsOnMars: ~/Work/shell\_exam' showing the execution of a script. The prompt is 'kbpark@WindowsOnMars:shell\_exam\$'. The user enters 'bash ./while\_test.sh'. The script outputs 'Number: 0', 'Number: 1', and 'Number: 2' on separate lines. The prompt then returns to 'kbpark@WindowsOnMars:shell\_exam\$' with a cursor.

```
kbpark@WindowsOnMars: ~/Work/shell_exam
kbpark@WindowsOnMars:shell_exam$ bash ./while_test.sh
Number: 0
Number: 1
Number: 2
kbpark@WindowsOnMars:shell_exam$
```

# LINUX - SHELL SCRIPT

## 날짜를 이용한 루프

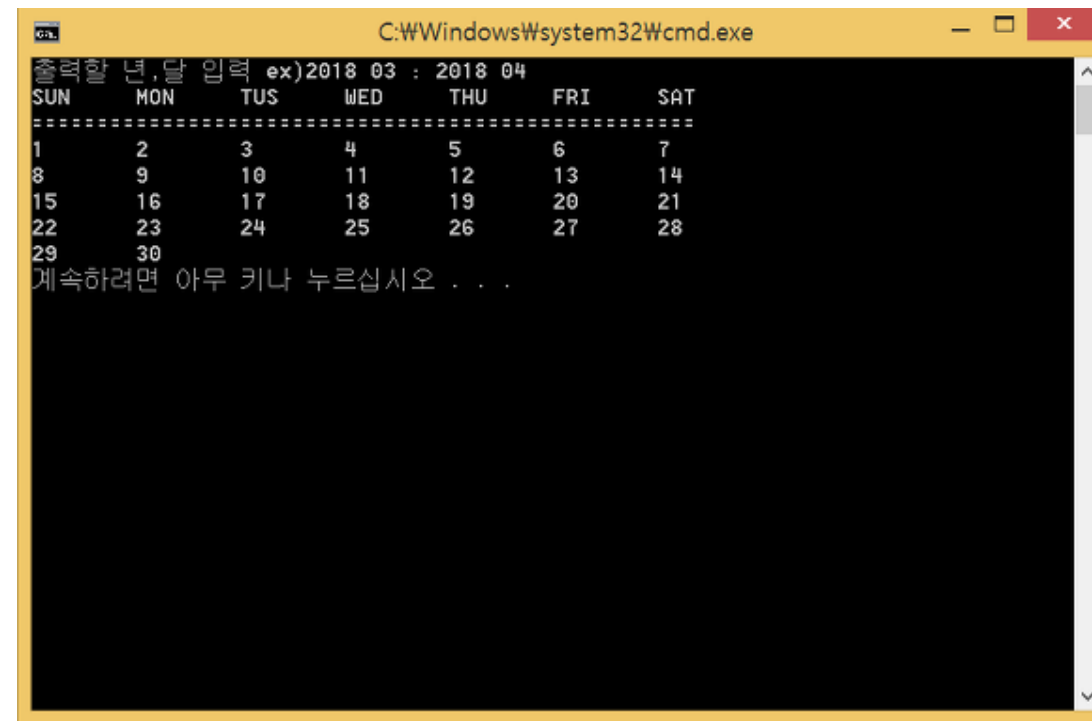
- `#!/bin/bash`
- `startDate=`date +%Y%m%d" -d "20210101```
- `endDate=`date +%Y%m%d" -d "20210201```
- `while [ "$startDate" != "$endDate" ] ;`
- `do`
- `echo $startDate`
- `startDate=`date +%Y%m%d" -d "$startDate + 1 day";`
- `done`



```
kbpark@WindowsOnMars: ~/Work/shell_exam
20210103
20210104
20210105
20210106
20210107
20210108
20210109
20210110
20210111
20210112
20210113
20210114
20210115
20210116
20210117
20210118
20210119
20210120
20210121
20210122
20210123
20210124
20210125
20210126
20210127
20210128
20210129
20210130
20210131
kbpark@WindowsOnMars:shell_exam$
```

# 과제

- 년도와 월을 입력 받아 달력을 출력 한다.
- 윤년 규칙
  - 1) 4년으로 나누어지는 해는 윤달(2월은 29일)이 있다.
  - 2) 그러나 100년으로 나누어지는 해는 윤달이 없다.
  - 3) 그러나 400년으로 나누어지는 해는 윤달이 있다. 이 규칙을 만족해야만 그 해가 윤년(2월은 29일)이고
  - 아닌 경우에는 평년(2월은 28일)이다.



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The prompt shows a command to display a calendar for March 2018: `ex) 2018 03 : 2018 04`. The output is a text-based calendar grid. The days of the week are listed as SUN, MON, TUS, WED, THU, FRI, SAT. The dates 1 through 30 are arranged in a grid. The 29th is shown as a Sunday, indicating it is a leap year. Below the grid, there is a prompt: `계속하려면 아무 키나 누르십시오 . . .`.

SUN	MON	TUS	WED	THU	FRI	SAT
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					