

# 리눅스 셸 프로그래밍

## 기본부분

### 스크립트 구축의 기초

- 교육내용
  - 여러 명령 사용하기
  - 스크립트 파일 작성하기
  - 메시지 표시하기
  - 변수 사용하기
  - 계산 수행하기

## 여러 명령 사용하기

- 두 개의 명령을 함께 실행하고 싶다면 같은 프롬프트 라인에 세미콜론(;) 으로 둘을 구분하여 입력

```
[root@srv1 ~]# date ; who
Tue Dec 17 15:55:50 KST 2019
root      :0      2019-12-09 15:05 (:0)
root      pts/1    2019-12-11 14:10 (:0)
root      pts/2    2019-12-11 17:00 (:0)
root      pts/3    2019-12-17 11:59 (:0)
[root@srv1 ~]# █
```

- 한 줄에 입력할 수 있는 명령의 한계는 최대 255 글자
- 이 명령들을 실행할 때마다 커맨드 프롬프트에 전체 명령을 입력해야 한다 > 이러한 명령을 간단한 텍스트 파일에 넣어 둘 수 있다

## 스크립트 파일 만들기

- 셸 스크립트 파일을 만들 때 첫번째 줄에는 사용하고 있는 셸을 지정한다

```
#!/bin/bash
```

- 셸을 표시한 후 파일의 각 줄에 명령을 입력하고 줄 바꿈. “#” 은 주석을 의미한다

```
#!/bin/bash
# This script displays the date and who's logged on
date
who
```

셸은 파일에 나타나는 순서대로 명령을 처리

- chmod 를 사용하여 권한을 조정하고 아래와 같이 실행

```
# chmod u+x test1
# ./test1
```

## 메시지 표시하기

- 스크립트 안에서 별도의 문자 메시지를 추가하고 싶다면 echo 명령을 사용할 수 있다.

```
[root@srv1 ~]# echo This is a test
This is a test
[root@srv1 ~]#
[root@srv1 ~]# echo Let's see if this'll work
Let's see if this'll work
[root@srv1 ~]#
[root@srv1 ~]# echo "Let's see if this'll work"
Let's see if this'll work
[root@srv1 ~]#
[root@srv1 ~]# █
```

- 명령의 출력과 같은 줄에 텍스트 문자열을 표기하고 싶다면?

```
[root@srv1 1217]# cat test1
#!/bin/bash

echo "The time and date are: "
date
echo
echo -n "The time and date are: "
date
[root@srv1 1217]# ./test1
The time and date are:
Tue Dec 17 16:11:42 KST 2019

The time and date are: Tue Dec 17 16:11:42 KST 2019
[root@srv1 1217]#
```

## 변수 사용하기

- 정보를 처리하기 위해 쉘 명령에 다른 데이터를 넣고 싶다면 변수를 이용할 수 있다. 변수는 쉘 스크립트 안에서 임시로 정보를 저장했다가 스크립트 안의 다른 명령에서 활용할 수 있다.

- 환경변수

- set 명령을 사용하면 사용할 수 있는 환경 변수의 전체 목록을 표시할 수 있다
- 환경 변수 이름 앞에 \$ 를 사용하면 스크립트 안에서 환경 변수를 활용할 수 있다

```
[root@srv1 1217]# cat test2
#!/bin/bash

echo "User info for userid: $USER"
echo UID: $UID
echo HOME: $HOME
[root@srv1 1217]# ./test2
User info for userid: root
UID: 0
HOME: /root
[root@srv1 1217]#
```

- 스크립트 안에서 \$ 를 표시하고 싶다면 \$\$ 와 같이 표시해야 한다
- \${변수} 형식으로도 변수 참조가 가능하다

## 변수 사용하기 (Cont.)

- 사용자 변수

환경 변수 말고도 쉘 스크립트는 스크립트 안에 자체 변수를 설정하고 사용할 수 있다. 변수를 설정하면 데이터를 임시로 저장하고 스크립트 안에서 사용할 수 있으며, 이는 쉘 스크립트를 좀 더 실제 컴퓨터 프로그램과 같이 만드는데 도움이 된다

사용자 변수는 최대 20 글자로 숫자 또는 밑줄로 이루어진 텍스트 문자열이 될 수 있다. 사용자 변수는 대소문자를 구분하므로 변수 Var1 과 var1 은 다르다.

값은 등호를 사용하여 사용자 변수에 할당된다. 변수, 등호, 값 사이에는 빈 칸을 둘 수 없다. 다음은 변수에 값을 할당하는 몇 가지 예다

```
var1=10
var2=-57
var3=testing
var4="still more testing"
```

## 변수 사용하기 (Cont.)

- 사용자 변수

```
[root@srv1 1217]# cat test3
#!/bin/bash

days=10
guest="Katie"
echo "$guest checked in $days days ago"

days=5
guest="Jessica"
echo "$guest checked in $days days ago"
[root@srv1 1217]#
[root@srv1 1217]# ./test3
Katie checked in 10 days ago
Jessica checked in 5 days ago
[root@srv1 1217]# █
```

- 변수를 언급할 때마다 현재 할당된 값을 돌려준다

```
[root@srv1 1217]# cat test4
#!/bin/bash

value1=10
value2=$value1
echo "The resulting value is $value2"
[root@srv1 1217]# ./test4
The resulting value is 10
[root@srv1 1217]#
```

## 명령 치환하기

- 명령의 출력으로부터 정보를 추출하고 이를 변수에 할당할 수 있는 기능
  - 역따옴표 문자 ( ` ` )      예) testing=`date`
  - \$( ) 형식                      예) testing=\$(date)

```
[root@srv1 1217]# cat test5
#!/bin/bash

today=$(date +%y%m%d)
ls /usr/bin -al > log.$today
[root@srv1 1217]#
[root@srv1 1217]# ./test5
[root@srv1 1217]#
[root@srv1 1217]# ls log.*
log.191217
[root@srv1 1217]# █
```

## 계산하기

- 셸 스크립트에서 수학 연산을 수행하는 두 가지 방법

expr 명령	\$ expr 1 + 5
대괄호 사용하기	\$ var1=\$((1 + 5))

```
[root@srv1 shell]# expr 5 * 2
10
[root@srv1 shell]# expr 5 \* 2
10
[root@srv1 shell]# expr 10 / 2
5
[root@srv1 shell]# █
[root@srv1 shell]# cat test6
#!/bin/bash

var1=10
var2=20
var3=$((expr $var1 / $var2))
echo The result is $var3
[root@srv1 shell]#
[root@srv1 shell]# ./test6
The result is 0
[root@srv1 shell]#
```

```
[root@srv1 shell]# cat test7
#!/bin/bash

var1=100
var2=50
var3=45
var4=$((var1 * (var2 - var3)))
echo The final result is $var4
[root@srv1 shell]#
[root@srv1 shell]# ./test7
The final result is 500
[root@srv1 shell]#
```

## 계산하기 (Cont.)

- expr 명령 연산자

연산자	설명	연산자	설명
ARG1   ARG2	어느 매개변수도 null 또는 0 이 아니면 ARG1을, 그렇지 않으면 ARG2를 돌려준다	ARG1 % ARG2	ARG1을 ARG2로 나눈 나머지를 돌려준다
ARG1 & ARG2	어느 매개변수도 null 또는 0이 아니면 ARG1을, 그렇지 않으면 0을 돌려준다	STRING : REGEXP	STRING 문자열 안에 정규표현식 REGEXT와 일치하는 패턴이 있다면 STRING 안에 있는 패턴을 돌려준다
ARG1 < ARG2	ARG1이 ARG2보다 작으면 1을, 그렇지 않으면 0을 돌려준다	match STRING REGEXP	STRING 문자열 안에 정규표현식 REGEXT와 일치하는 패턴이 있다면 STRING 안에 있는 패턴을 돌려준다
ARG1 <= ARG2	ARG1이 ARG2보다 작거나 같으면 1을, 그렇지 않으면 0을 돌려준다	substr STRING POS LENGTH	STRING 문자열 안에서 POS(1로 시작) 위치에서 시작하는 LENGTH 길이만큼의 문자열을 돌려준다
ARG1 != ARG2	ARG1과 ARG2가 같지 않으면 1을, 그렇지 않으면 0을 돌려준다	index STRING CHARS	STRING 문자열에서 CHARS가 발견된 위치를 돌려주면 없으면 0을 돌려준다
ARG1 >= ARG2	ARG1이 ARG2 보다 크거나 같으면 1을, 그렇지 않으면 0을 돌려준다	length STRING	STRING 문자열의 길이를 숫자로 돌려준다
ARG1 > ARG2	ARG1이 ARG2 보다 크면 1을, 그렇지 않으면 0을 돌려준다	+ TOKEN	TOKEN을 문자열로 변환한다. 키워드라고 해도 마찬가지다
ARG1 + ARG2	ARG1과 ARG2의 덧셈 결과를 돌려준다	(EXPRESSION)	EXPRESSION 식의 값을 돌려준다
ARG1 - ARG2	ARG1과 ARG2의 뺄셈 결과를 돌려준다		
ARG1 * ARG2	ARG1과 ARG2의 곱셈 결과를 돌려준다		
ARG1 / ARG2	ARG1을 ARG2로 나눈 몫을 돌려준다		

## 부동소수점을 위한 해법

- bash가 정수 연산만 할 수 있는 한계를 극복하기 위한 대표적인 방법은 내장된 bash 계산기인 bc 를 사용하는 것이다
- bc 의 기초  
bash 계산기는 다음을 인식한다

숫자 (정수 및 부동 소수점 모두)  
변수 (단순 변수와 배열 모두)  
주석 (# 기호로 시작하는 줄 또는 C 언어의 /\* \*/ 쌍으로 된 주석)  
수식  
프로그래밍문(예, if-then)  
함수

```
[root@srv1 test]# bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
12 * 1.5
18.0

3.14 * ( 4 + 5 )
28.26
quit
[root@srv1 test]#
```

## 부동소수점을 위한 해법 (Cont.)

- 부동 소수점 계산은 scale 이라는 내장변수가 제어한다. 이 값에 원하는 소수점 이하 자릿수를 설정하지 않으면 원하는 답을 찾지 못할 수 있다. scale 변수의 기본값은 '0'

```
[root@srv1 test]# bc -q
5/4
1

scale=5
5/4
1.25000

quit
[root@srv1 test]#
```

- 일반 숫자에 더해서 bash 계산기는 변수도 이해한다

```
[root@srv1 test]# bc -q
var1=10
var1 * 4
40

var2 = var1 / 5
print var2
2
```

## 부동소수점을 위한 해법 (Cont.)

- 스크립트에서 bc 사용하기  
bc 명령을 실행하고 출력을 변수에 할당하는 명령 치환 기능을 사용할 수 있다. 기본 형식은 아래와 같다

variable=\$(echo "options; expression" | bc)

변수설정, 두 개 이상의 변수 설정해야 할 때에는  
세미콜론으로 구분

expression 매개변수는 BC로 계산할 수식을 정의

```
[root@srv1 test]# cat test9
#!/bin/bash

var1=$(echo "scale=4; 3.44 / 5" | bc)
echo the answer is $var1
[root@srv1 test]# ./test9
the answer is .6880
[root@srv1 test]# cat test10
#!/bin/bash
var1=100
var2=45
var3=$(echo "scale=4; $var1 / $var2" | bc)
echo "The answer for this is $var3"
[root@srv1 test]# ./test10
The answer for this is 2.2222
[root@srv1 test]#
```

## 구조적 명령 사용하기

- 교육내용
  - if-then 구문 사용하기
  - 중첩된 if 문
  - test 명령 이해하기
  - 복합 조건 테스트하기
  - 이중 대괄호 및 괄호 사용하기
  - case 구문보기

## if-then 구문 사용하기

- 구조적 명령의 가장 기본적인 유형은 if-else 구문이다. if-else 구문의 형식은 다음과 같다

```
if command
then
    commands
fi
```

다른 프로그래밍 언어에서는 if 문 다음에 나오는 객체는 TRUE 또는 FALSE 값을 평가하는 표현식이다. bash 셸은 if 문은 그 줄에 정의된 명령을 실행한다. 이 명령의 종료 상태가 0(명령이 성공적으로 완료됨) 이라면 then 아래에 있는 명령이 실행된다. 명령이 종료 상태가 아니라면 then 아래에 있는 명령은 실행되지 않고, bash 셸은 스크립트의 다음 명령으로 넘어간다. fi 문은 the if-then 구문의 끝을 의미한다.

```
[root@srv1 test]# cat test1.sh
#!/bin/bash

if pwd
then
    echo "It worked"
fi

[root@srv1 test]# ./test1.sh
/root/test
It worked
[root@srv1 test]# █
```



## if-then 구문 사용하기 (Cont.)

- 아래와 같이 잘못된 명령인 경우 0이 아닌 종료 상태를 만들어 내고 bash 셸은 then 섹션의 echo 문을 생각한다. 또한 if 문에서 실행되는 명령의 오류 메시지가 스크립트의 출력에 나타난다

```
[root@srv1 test]# cat test2.sh
#!/bin/bash
if !amNotaCommand
then
    echo "It worked"
fi
echo "We are outside the if statement"
[root@srv1 test]# ./test2.sh
./test2.sh: line 2: !amNotaCommand: command not found
We are outside the if statement
[root@srv1 test]#
```

## if-then 구문 사용하기 (Cont.)

- then 섹션에는 여러 명령을 넣을 수 있다. bash 는 이 명령들을 한 블록으로 다룬다. 명령이 0의 종료 상태를 돌려주면 이 명령들을 모두 처리하며 명령이 0이 아닌 종료 상태를 돌려주면 모두 건너뛰는다

```
[root@srv1 test]# cat test3.sh
#!/bin/bash
testuser=brian
if grep $testuser /etc/passwd
then
    echo "This is my first command"
    echo "This is my second command"
    echo "I can even put in other commands besides echo:"
    ls -a /home/$testuser/.b*
fi
[root@srv1 test]# ./test3.sh
[root@srv1 test]# sed -i 's/brian/user1/g' test3.sh
[root@srv1 test]# ./test3.sh
user1:x:1000:1000:user1:/home/user1:/bin/bash
This is my first command
This is my second command
I can even put in other commands besides echo:
/home/user1/.bash_history /home/user1/.bash_profile
/home/user1/.bash_logout /home/user1/.bashrc
[root@srv1 test]#
```

## if-then 구문 들여다보기

- if-then 구문에서 명령이 0이 아닌 종료 상태코드를 돌려주면 bash 셸은 스크립트의 다음 명령으로 넘어간다. 이러한 상황에서 다른 명령 세트를 실행할 수 있다. 이것이 if-then-else 구문이다

```
[root@srv1 test]# cat test4.sh
#!/bin/bash

testuser=NoSuchUser
if grep $testuser /etc/passwd
then
    echo "The bash files for user $testuser are:"
    ls -a /home/$testuser/.b*
    echo
else
    echo "The user $testuser does not exist on this system"
    echo
fi
[root@srv1 test]# ./test4.sh
The user NoSuchUser does not exist on this system

[root@srv1 test]#
```

## 중첩된 if 문

- 중첩된 if 문을 쓰면 스크립트 코드의 상황을 확인할 수 있다. 로그인 이름이 /etc/passwd 파일에 없지만 그 사용자의 디렉토리가 아직 남아 있는지 확인하려면 중첩된 if-else 구문을 사용한다. 이런 경우 중첩된 if-else 구문은 메인 if-then-else 구문의 else 블록안에 놓인다

```
[root@srv1 test]# cat test5.sh
#!/bin/bash

testuser=NoSuchUser

if grep $testuser /etc/passwd
then
    echo "The user $testuser exists on this system"
else
    echo "The user $testuser does not exist on this system"
    if ls -d /home/$testuser/
    then
        echo "However, $testuser has a directory"
    fi
fi
[root@srv1 test]# ./test5.sh
The user NoSuchUser does not exist on this system
ls: cannot access /home/NoSuchUser/: No such file or directory
[root@srv1 test]#
[root@srv1 test]#
[root@srv1 test]# useradd NoSuchUser
[root@srv1 test]# passwd NoSuchUser
Changing password for user NoSuchUser.
New password:
BAD PASSWORD: The password is shorter than 8 characters
Retype new password:
passwd: all authentication tokens updated successfully.
[root@srv1 test]# userdel NoSuchUser
[root@srv1 test]# ./test5.sh
The user NoSuchUser does not exist on this system
/home/NoSuchUser/
However, NoSuchUser has a directory
[root@srv1 test]#
```

## 중첩된 if 문 (Cont.)

- if-then 구문을 별도로 쓰는 대신 else 부분의 다른 버전인 elif 문을 이용할 수 있다. elif 는 if-then 구문의 else 부분을 이어 나간다.

```
if command1
then
    commands
elif command2
then
    more commands
fi
```

elif 문의 줄은 원래의 if 문의 줄과 비슷하게 다른 명령을 평가할 수 있다. elif 명령의 종료 상태 코드가 0 이라면 bash 는 두 번째 then 문 부분의(more commands)을 실행한다. 이런 방식의 구문 중첩은 코드를 더욱 깔끔하게 만들고 논리 흐름을 더 따라가기 쉽게 만들어 준다

## 중첩된 if 문 (Cont.)

```
[root@srv1 test]# cat test5.sh
#!/bin/bash

testuser=NoSuchUser

if grep $testuser /etc/passwd
then
    echo "The user $testuser exists on this system"

elif ls -d /home/$testuser
then
    echo "The user $testuser does not exist on this system"
    echo "However, $testuser has a directory"
fi
[root@srv1 test]# ./test5.sh
/home/NoSuchUser
The user NoSuchUser does not exist on this system
However, NoSuchUser has a directory
[root@srv1 test]#
```

- 위의 스크립트를 진전시켜 디렉토리에 남아 있지만 존재하지 않는 사용자와 디렉토리도 남아있지 않은 존재하지 않는 사용자 모두를 확인할 수 있다. elif 문 안에 else 문을 추가하면 된다

## 중첩된 if 문 (Cont.)

```
[root@srv1 test]# cat test5.sh
#!/bin/bash

testuser=NoSuchUser

if grep $testuser /etc/passwd
then
    echo "The user $testuser exists on this system"

elif ls -d /home/$testuser
then
    echo "The user $testuser does not exist on this system"
    echo "However, $testuser has a directory"

else
    echo "The user $testuser does not exist on this system"
    echo "And, $testuser does not have a directory"
fi
[root@srv1 test]# ./test5.sh
/home/NoSuchUser
The user NoSuchUser does not exist on this system
However, NoSuchUser has a directory
[root@srv1 test]# rm -rf /home/NoSuchUser/
[root@srv1 test]#
[root@srv1 test]# ./test5.sh
ls: cannot access /home/NoSuchUser: No such file or directory
The user NoSuchUser does not exist on this system
And, NoSuchUser does not have a directory
[root@srv1 test]#
```

## 중첩된 if 문 (Cont.)

- 계속해서 elif 문을 이어 써서, 하나의 커다란 if-then-elif 복합체를 만들 수 있다

```
if command1
then
    command set 1
elif command2
then
    command set 2
elif command3
then
    command set 3
elif command4
then
    command set 4
fi
```

- 각 명령 세트는 명령이 종료 상태 코드로 0을 돌려주는지 여부에 따라서 실행된다. bash 셸은 if 문을 순서대로 실행하면 첫 번째 if 문이 종료 코드로 0을 돌려줄 때에만 then 섹션이 실행된다

## 테스트 명령 써 보기

- 테스트 명령은 if-then 구문에서 여러 가지 조건을 테스트하는 방법을 제공한다. 테스트 명령에 나와 있는 조건이 참으로 평가되면 테스트 명령은 종료 상태 코드로 0을 돌려준다. 이를 활용하면 if-then 구문이 다른 프로그래밍 언어의 if-then 과 비슷하게 동작한다. 조건이 거짓이면 test 명령은 0이 아닌 상태 코드를 돌려주므로 if-then 구문에서 빠져 나가게 된다

테스트 명령의 형식은 다음과 같이 매우 간단하다

```
test condition
```

condition 은 테스트 명령이 평가할 일련의 명령 및 매개변수다. if-then 구문에 사용하는 테스트 명령은 다음과 같다.

```
if test condition
then
    명령
fi
```

테스트 명령문의 condition 부분을 비워 놓으면 0이 아닌 종료 상태 코드로 종료되므로 else 블록문으로 넘어간다

## 테스트 명령 써 보기 (Cont.)

```
[root@srv1 test]# cat test6.sh
#!/bin/bash

if test
then
    echo "No expression returns a True"
else
    echo "No expression returns a False"
fi
[root@srv1 test]# ./test6.sh
No expression returns a False
[root@srv1 test]#
```

- 조건을 추가하면 테스트 명령이 이를 테스트한다. 예를 들어, 테스트 명령을 사용하여 어떤 변수가 내용을 가지고 있는지를 판단할 수 있다. 변수가 내용을 가지고 있는지 확인하려면 간단한 조건식이 필요하다

```
[root@srv1 test]# cat test6.sh
#!/bin/bash

my_variable="Full"

if test $my_variable
then
    echo "The $my_variable expression returns a True"
else
    echo "The $my_variable expression returns a False"
fi
[root@srv1 test]# ./test6.sh
The Full expression returns a True
[root@srv1 test]#
```

## 테스트 명령 써 보기 (Cont.)

- bash 쉘은 if-then 구문에서 테스트 명령을 쓰지 않고도 조건을 테스트하는 다른 방법을 제공한다

```
if [ condition ]
then
    명령
fi
```

- 대괄호는 테스트 조건을 정의한다. 조건과 괄호사이에는 각각 빈칸이 있어야 한다
- 테스트 명령 및 테스트 조건은 세 가지 종류의 조건을 평가할 수 있다
  - ▶ 숫자 비교
  - ▶ 문자열 비교
  - ▶ 파일 비교

## 테스트 명령 써 보기 (Cont.)

- 숫자 비교 사용하기

비교	설명
<code>n1 -eq n2</code>	n1 과 n2가 같은 지 검사한다
<code>n1 -ge n2</code>	n1 이 n2보다 크거나 같은 지 검사한다
<code>n1 -gt n2</code>	n1 이 n2보다 큰 지 검사한다.
<code>n1 -le n2</code>	n1 이 n2보다 작거나 같은 지 검사한다.
<code>n1 -lt n2</code>	n1 이 n2보다 작은 지 검사한다.
<code>n1 -ne n2</code>	n1 과 n2가 같지 않은 지 검사한다

- 숫자 비교는 숫자 및 변수를 평가할 때 모두 사용될 수 있다. 단, 부동소수점은 사용할 수 없다

```
[root@srv1 test]# cat numeric_test.sh
#!/bin/bash

value1=10
value2=11

if [ $value1 -gt 5 ]
then
    echo "The test value $value1 is grater then 5"
fi

if [ $value1 -eq $value2 ]
then
    echo "The values are equal"
else
    echo "values are different"
fi
[root@srv1 test]# ./numeric_test.sh
The test value 10 is grater then 5
values are different
[root@srv1 test]#
```

## 테스트 명령 써 보기 (Cont.)

- 문자열 비교 사용하기

비교	설명
<code>str1 = str2</code>	str1 이 str2와 같은 지 검사한다
<code>str1 != str2</code>	str1 이 str2 와 같지 않은 지 검사한다
<code>str1 &lt; str2</code>	str1 이 str2 보다 작은 지 검사한다
<code>str1 &gt; str2</code>	str1 이 str2 보다 큰 지 검사한다
<code>-n str1</code>	str1 의 길이가 0 보다 큰 지 검사한다
<code>-z str1</code>	str1 의 길이가 0인지 검사한다

- 문자열이 일치하는 지 보기

```
[root@srv1 test]# cat test7.sh
#!/bin/bash

testuser=rich

if [ $USER = $testuser ]
then
    echo "Welcome $testuser"
fi
[root@srv1 test]# ./test7.sh
[root@srv1 test]#
```

## 테스트 명령 써 보기 (Cont.)

- 문자열 비교 사용하기

- 문자열이 같지 않은 지 비교해도 두 문자열이 같은 값인지 아닌지를 판단할 수 있다

```
[root@srv1 test]# cat test8.sh
#!/bin/bash

testuser=baduser

if [ $USER != $testuser ]
then
    echo "This is not $testuser"
else
    echo "Welcome $testuser"
fi
[root@srv1 test]# ./test8.sh
This is not baduser
[root@srv1 test]#
```

- 문자열이 같은 지 비교할 때에는 모든 문장부호와 대문자도 고려된다.

## 테스트 명령 써 보기 (Cont.)

- 문자열 비교 사용하기

- 문자열의 크고 작음을 보기
- 문자열의 크기 여부를 테스트하는 기능을 사용하려고 할 때는 다음의 두 가지를 고려해야 한다

부등호 기호를 이스케이프 해야 함. 그렇지 않으면 쉘은 이를 리다이렉트로 해석해서 문자열 값을 파일 이름으로 사용함  
어느 것이 더 큰지 순서를 결정하는 논리는 sort 명령에서 쓰이는 것과 같지 않음

```
[root@srv1 test]# cat test9.sh
#!/bin/bash

val1=baseball
val2=hockey

if [ $val1 \> $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
[root@srv1 test]# ./test9.sh
baseball is less than hockey
[root@srv1 test]#
```

## 테스트 명령 써 보기 (Cont.)

- 문자열 크기 보기

- n 및 -z 비교는 변수에 데이터가 포함되어 있는지 여부를 평가하려고 할 때 편리

```
[root@storage shell]# cat test10.sh
#!/bin/bash

val1=testing
val2=''

if [ -n $val1 ]
then
    echo "The string '$val1' is not empty"
else
    echo "The string '$val1' is empty"
fi

if [ -z $val2 ]
then
    echo "The string '$val2' is empty"
else
    echo "The string '$val2' is not empty"
fi

if [ -z $val3 ]
then
    echo "The string '$val3' is empty"
else
    echo "The string '$val3' is not empty"
fi
[root@storage shell]# ./test10.sh
The string 'testing' is not empty
The string '' is empty
The string '' is empty
[root@storage shell]#
```



## 테스트 명령 써 보기 (Cont.)

- 파일 비교 사용하기

- 파일비교는 쉘 스크립트에서 가장 많이 사용되는 비교이다

비교	설명
-d file	파일이 존재하고 디렉토리인지 검사한다
-e file	파일이 존재하는지 검사한다
-f file	파일이 존재하고 파일인지 검사한다
-r file	파일이 존재하고 읽을 수 있는 지 검사한다
-s file	파일이 존재하고 비어 있지 않은 지 검사한다
-w file	파일이 존재하고 기록할 수 있는 지 검사한다
-x file	파일이 존재하고 실행할 수 있는 지 검사한다
-O file	파일이 존재하고 현재 사용자가 소유한 것인지 검사한다
-G file	파일이 존재하고 기본 그룹이 현재 사용자와 같은 지 검사한다
file1 -nt file2	file1 이 file2보다 새것인지 검사한다
file1 -ot file2	file2이 file2보다 오래된 것인지 검사한다

## 테스트 명령 써 보기 (Cont.)

- 파일 비교 사용하기

- 디렉토리 확인하기  
지정된 디렉토리가 시스템에 존재하는지 보려면 -d 검사를 한다. 보통은 디렉토리에 파일을 쓰거나 디렉토리의 위치를 변경하기 전에 사용하면 좋다

```
[root@storage shell]# cat test11.sh
#!/bin/bash

jump_directory=/etc/test

if [ -d $jump_directory ]
then
    echo "The $jump_directory directory exists"
    cd $jump_directory
    ls
else
    echo "The $jump_directory directory does not exist"
fi
[root@storage shell]# ./test11.sh
The /etc/test directory does not exist
[root@storage shell]#
[root@storage shell]# mkdir /etc/test
[root@storage shell]#
[root@storage shell]# ./test11.sh
The /etc/test directory exists
[root@storage shell]#
[root@storage shell]# touch /etc/test/1.txt
[root@storage shell]#
[root@storage shell]# ./test11.sh
The /etc/test directory exists
1.txt
[root@storage shell]# █
```

## 테스트 명령 써 보기 (Cont.)

### • 파일 비교 사용하기

- 개체가 존재하는지 여부 확인하기  
스크립트에서 파일 또는 디렉토리를 사용하기 전에 이 개체가 있는지 확인하려면 -e 비교를 사용한다

```
[root@storage shell]# cat test12.sh
#!/bin/bash

location=$HOME
file_name="sentinel"

if [ -e $location ]
then
    echo "OK on the $location directory"
    echo "Now checking on the file, $file_name."

    if [ -e $location ]
    then
        echo "OK on the filename"
        echo "Updating Current Data..."
        date >> $location/$file_name
    else
        echo "File does not exist"
        echo "Nothing to update"
    fi
else
    echo "The $location directory does not exist"
    echo "Nothing to update"
fi

[root@storage shell]# ./test12.sh
OK on the /root directory
Now checking on the file, sentinel.
OK on the filename
Updating Current Data...
[root@storage shell]# touch $HOME/sentinel
[root@storage shell]# ./test12.sh
OK on the /root directory
Now checking on the file, sentinel.
OK on the filename
Updating Current Data...
```

## 테스트 명령 써 보기 (Cont.)

### • 파일 날짜 확인하기

- 두 파일이 만들어진 시간을 대상. 이는 소프트웨어를 설치하는 스크립트를 작성할 때 편리하다. 이미 시스템에 설치되어 있는 파일보다 오래된 파일을 설치하지 않아야 할 때가 있다. -nt 비교는 어떤 파일이 다른 파일보다 최신인지 여부를 확인할 수 있다

```
[root@storage shell]# cat test20.sh
#!/bin/bash

if [ test12.sh -nt test11.sh ]
then
    echo "The test12 file is newer than test11"
else
    echo "The test11 file is newer than test12"
fi

if [ test10.sh -ot test11.sh ]
then
    echo "The test10 file is older than the test11 file"
fi

[root@storage shell]# ./test20.sh
The test12 file is newer than test11
The test10 file is older than the test11 file
[root@storage shell]# █
```

위의 예는 실행 위치에 따라 문제가 발생할 수 있으므로 사전에 파일의 존재 여부를 확인해야 한다.

## 복합 테스트 검토하기

- if-then 구문을 사용하면 테스트를 결합하는 부울 논리(Boolean logic)를 사용할 수 있으며 아래와 같이 두 가지 부울 연산자 사용이 가능하다

- [ condition1 ] && [ condition2 ] (AND 부울 연산자)
- [ condition1 ] || [ condition2 ] (OR 부울 연산자)

부울 논리는 가능한 반환값이 TRUE 또는 FALSE로 한정된다

```
[root@storage shell]# cat test22.sh
#!/bin/bash

if [ -d $HOME ] && [ -w $HOME/testing ]
then
    echo "The file exists and you can write to it"
else
    echo "I can't write to the file"
fi
[root@storage shell]#
[root@storage shell]# ./test22.sh
I can't write to the file
[root@storage shell]#
[root@storage shell]# touch $HOME/testing
[root@storage shell]#
[root@storage shell]# ./test22.sh
The file exists and you can write to it
[root@storage shell]#
[root@storage shell]#
```

## 고급 if-then 기능 사용하기

- if-then 구문에서는 아래와 같은 두 가지 추가 기능을 제공한다

- 수학 표현식을 위한 이중 괄호
- 고급 문자열 처리 기능을 위한 이중 대괄호

- 이중 괄호 사용하기

- 이중 괄호 명령은 비교에 고급 수학 공식을 통합할 수 있다. 테스트 명령은 비교에서 간단한 산술 연산을 할 수 있다. 이중 괄호 명령은 다른 프로그래밍 언어를 사용하고 있는 프로그래머가 익숙하게 사용할 수 있는 더 많은 수학 기호를 제공한다. 이중 괄호의 형식은 다음과 같다

(( expression ))

express 에는 수식 또는 비교 표현식이 들어갈 수 있다. test 명령에서 쓰이는 표준 수학 연산자 이외에도 이중 괄호 명령 안에 들어갈 수 있는 추가 연산자 목록은 다음과 같다

기호	설명	기호	설명
val++	후위 증가	**	지수화
val--	후위 감소	<<	비트를 왼쪽으로 시프트
++val	전위 증가	>>	비트를 오른쪽으로 시프트
--val	전위 감소	&	비트 단위 부울 AND
!	논리 부정		비트 단위 부울 OR
~	비트 부정	&&	논리 AND
	논리 OR		

## 고급 if-then 기능 사용하기 (Cont.)

- 값을 할당하기 위한 스크립트의 일반 명령 뿐만 아니라, if 문에서도 이중 괄호 명령 사용 가능

```
[root@storage shell]# cat test23.sh
#!/bin/bash

val1=10

if (( $val1 ** 2 > 90 ))
then
    (( val2 = $val1 ** 2 ))
    echo "The square of $val1 is $val2"
fi
[root@storage shell]#
[root@storage shell]# ./test23.sh
The square of 10 is 100
[root@storage shell]#
```

- 이중 괄호 안의 수식 안에서는 부등호에 이스케이프가 필요 없는 것을 알 수 있다. 이는 이중 괄호 명령이 가진 또 다른 고급 기능이다.

## 고급 if-then 기능 사용하기 (Cont.)

- 이중 대괄호 사용하기

- 이중 대괄호 명령은 문자열 비교에 대한 고급 기능을 제공한다. 이중 대괄호 명령의 형식은 다음과 같다

```
[[ expression ]]
```

- 이중 대괄호는 bash 에서는 잘 동작하지만 모든 셸이 이중 대괄호를 지원하지는 않는다
- 패턴 일치에서는 문자열 값과 비교할 정규표현식을 정의할 수 있다

```
[root@storage shell]# cat test24.sh
#!/bin/bash

if [[ $USER == r* ]]
then
    echo "Hello $USER"
else
    echo "Sorry, I do not know you"
fi
[root@storage shell]#
[root@storage shell]# ./test24.sh
Hello root
[root@storage shell]#
```

- 이중 등호(==) 는 오른쪽에 있는 문자열(r\*)을 패턴으로 간주한다

## case 명령 알아보기

- case 로의 전환

- 변수의 값을 평가할 때 가능한 몇 가지 값 중 하나에 해당하는지 확인할 경우가 있다. 이러한 경우 if-then-else 를 길게 써야 하는데 같은 변수에 대해 계속해서 검사할 때 elif 문만을 쓰는 대신 case 를 사용할 수 있다
- case 문의 일반 적인 형태는 아래와 같다

```
case variable in
  pattern1 | pattern2) commands1;;
  pattern3) commands2;;
  *) default commands;;
esac
```

```
#!/bin/bash

case $USER in
  rich | barbara)
    echo "welcome, $USER"
    echo "Please enjoy your visit" ;;

  testing)
    echo "Special testing account" ;;

  jessica)
    echo "Do not forget to log off when you're done" ;;

  *)
    echo "Sorry, you are not allowed here" ;;
esac
```

## 구조적 명령 더 알아보기

- 교육내용

- for 문으로 되풀이하기
- until 문으로 되풀이 하기
- while 문 사용하기
- 루프 결합하기
- 루프 출력 리다이렉트하기

## for 명령

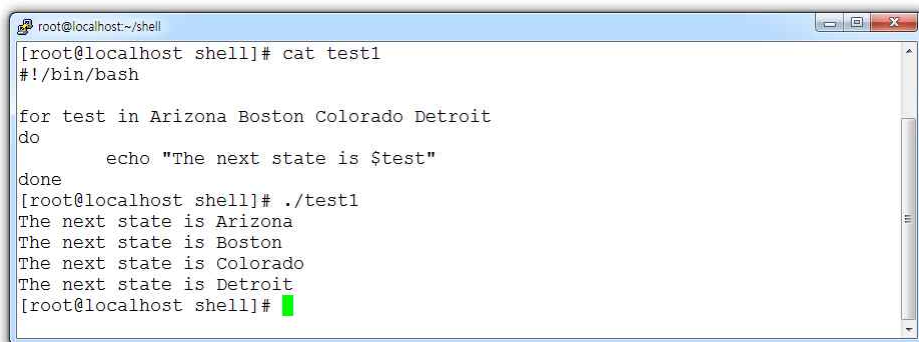
- 일련의 명령을 되풀이 하는 것은 프로그래밍에서 흔히 쓰이는 방법이다. 특정 조건이 만족될 때까지 일련의 명령을 되풀이 해야 한다는 것이다. 같은 디렉토리에 있는 모든 파일, 시스템에 있는 모든 사용자, 텍스트 파일의 모든 줄에 관한 작업을 할 때가 대표적인 예가 될 것이다
- bash 셸은 어떠한 집합을 거치면서 되풀이하는 루프 구문을 만들 수 있는 명령을 제공한다

```
for var in list
do
    commands
done
```

- 반복이 이루어질 때마다 var 변수는 list 중 현재의 값을 포함한다. 첫 번째 반복 단계에서는 list 의 첫 항목을 사용하고 두 번째 반복 단계에서는 두 번째 항목을 사용한다. 이런 식으로 list 안의 모든 항목을 사용할 때까지 계속 되풀이 된다

## for 명령 (Cont.)

- 목록에서 값을 읽기



```
root@localhost:~/shell
[root@localhost shell]# cat test1
#!/bin/bash

for test in Arizona Boston Colorado Detroit
do
    echo "The next state is $test"
done
[root@localhost shell]# ./test1
The next state is Arizona
The next state is Boston
The next state is Colorado
The next state is Detroit
[root@localhost shell]#
```

## for 명령 (Cont.)

- 목록에서 값을 읽기



```

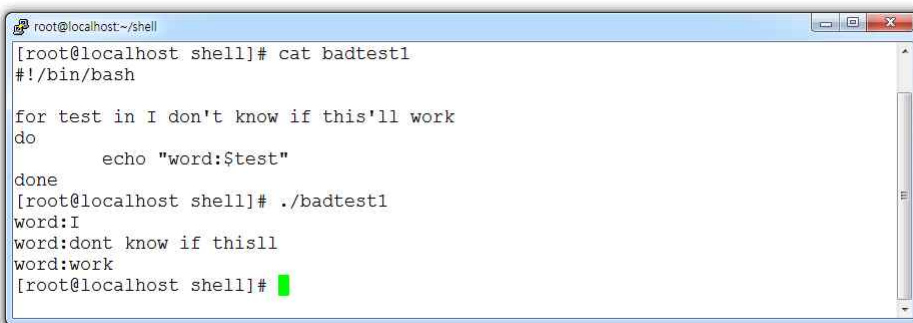
root@localhost:~/shell
[root@localhost shell]# cat test1b
#!/bin/bash

for test in Arizona Boston Colorado Detroit
do
    echo "The next state is $test"
done
echo "The last state we visited was $test"
test=NewYork
echo "Wait, now we're visiting $test"
[root@localhost shell]# ./test1b
The next state is Arizona
The next state is Boston
The next state is Colorado
The next state is Detroit
The last state we visited was Detroit
Wait, now we're visiting NewYork
[root@localhost shell]#
  
```

- \$test 변수는 쉘 스크립트의 나머지 부분에서 유효하며, 이 변수는 마지막 되풀이 할 때의 값을 유지한다.

## for 명령 (Cont.)

- 목록의 복잡한 값을 읽기  
다음은 쉘에서 발생할 수 있는 문제 중 대표적인 예이다



```

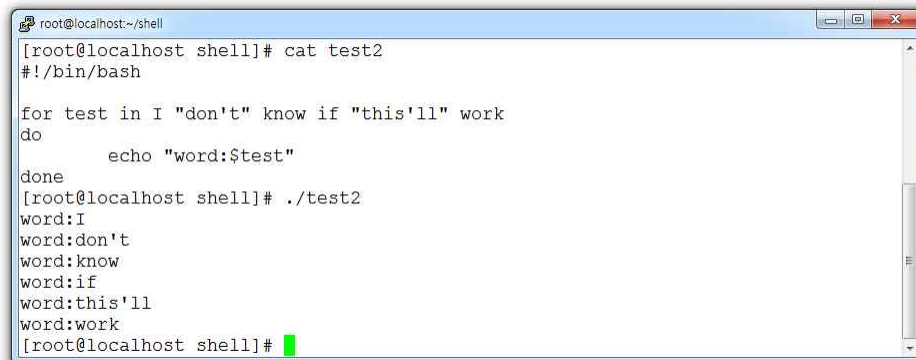
root@localhost:~/shell
[root@localhost shell]# cat badtest1
#!/bin/bash

for test in I don't know if this'll work
do
    echo "word:$test"
done
[root@localhost shell]# ./badtest1
word:I
word:don't know if this'll
word:work
[root@localhost shell]#
  
```

- 쉘은 목록 값 안에 있는 홑따옴표를 만나면 그 사이에 있는 값을 단일한 데이터 값으로 본다

## for 명령 (Cont.)

- 목록의 복잡한 값을 읽기
  - 홀따옴표를 이스케이프 문자(백슬래시)를 써서 이스케이프 처리
  - 홀따옴표를 사용하는 값을 정의하는 겹따옴표를 사용



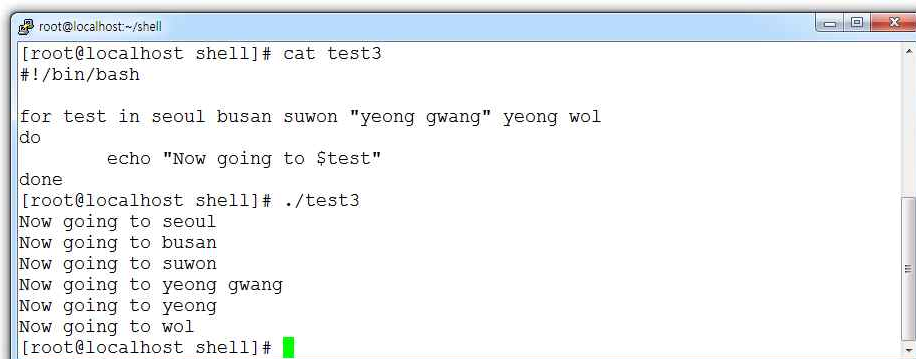
```

root@localhost:~/shell
[root@localhost shell]# cat test2
#!/bin/bash

for test in I "don't" know if "this'll" work
do
    echo "word:$test"
done
[root@localhost shell]# ./test2
word:I
word:don't
word:know
word:if
word:this'll
word:work
[root@localhost shell]#
  
```

## for 명령 (Cont.)

- 목록의 복잡한 값을 읽기
  - 개별 데이터 값 사이에 빈 칸이 있다면 이를 겹따옴표로 감싸야 한다



```

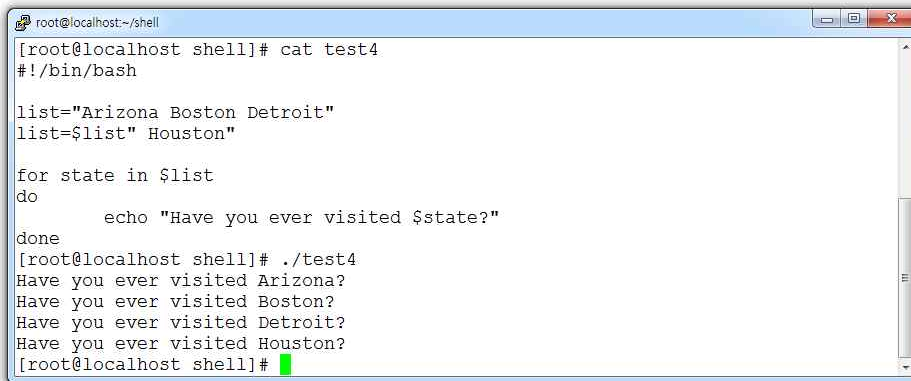
root@localhost:~/shell
[root@localhost shell]# cat test3
#!/bin/bash

for test in seoul busan suwon "yeong gwang" yeong wol
do
    echo "Now going to $test"
done
[root@localhost shell]# ./test3
Now going to seoul
Now going to busan
Now going to suwon
Now going to yeong gwang
Now going to yeong
Now going to wol
[root@localhost shell]#
  
```



## for 명령 (Cont.)

- 변수에서 목록 읽기  
셸 스크립트에서 발생한 값의 목록을 변수에 쌓아두었다가 되풀이해서 처리해야 할 때가 있다. for 명령으로 이러한 일도 할 수 있다



```

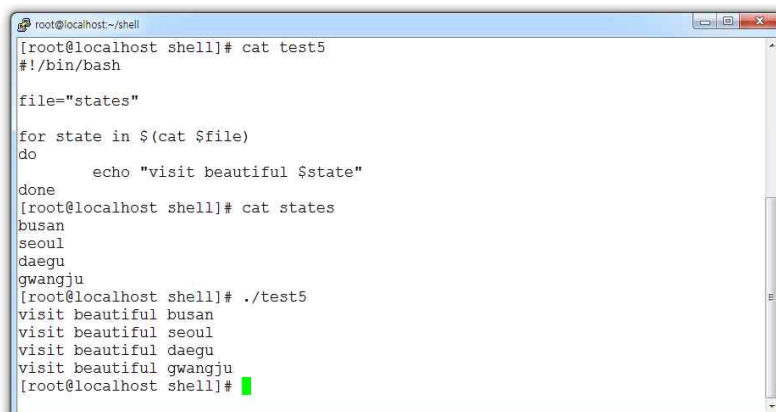
root@localhost:~/shell
[root@localhost shell]# cat test4
#!/bin/bash

list="Arizona Boston Detroit"
list=$list" Houston"

for state in $list
do
    echo "Have you ever visited $state?"
done
[root@localhost shell]# ./test4
Have you ever visited Arizona?
Have you ever visited Boston?
Have you ever visited Detroit?
Have you ever visited Houston?
[root@localhost shell]#
  
```

## for 명령 (Cont.)

- 명령에서 값을 읽기  
목록에 사용하기 위한 값을 만드는 또다른 방법은 명령의 출력을 사용하는 것이다. 사용자는 출력을 만들어 내는 어떤 명령이든 이를 실행하고, 그 출력을 for 명령에 사용하려면 명령을 치환한다



```

root@localhost:~/shell
[root@localhost shell]# cat test5
#!/bin/bash

file="states"

for state in $(cat $file)
do
    echo "visit beautiful $state"
done
[root@localhost shell]# cat states
busan
seoul
daegu
gwangju
[root@localhost shell]# ./test5
visit beautiful busan
visit beautiful seoul
visit beautiful daegu
visit beautiful gwangju
[root@localhost shell]#
  
```

## for 명령 (Cont.)

- 필드 구분자 변경하기  
내부 필드 분리 구분자(internal field separator)의 환경 변수가 현재는 한 번에 한줄씩 처리하도록 되어 있다. IFS 환경 변수는 bash 셸이 필드 구분자로 사용하는 문자의 목록을 정의한다. bash 셸은 기본적으로 다음 문자를 필드 구분을 위한 기호로 간주한다

- 빈 칸
- 탭
- 줄바꿈

bash 셸은 데이터에서 이러한 문자를 보게 되면 목록 안에서 새 데이터 필드가 시작된다고 본다. 이 문제를 해결하기 위해서는 bash 셸이 일시적으로 필드 구분자로 인식하는 문자를 제한할 목적으로 수레 스크립트에서 IFS 환경 변수 값을 바꿀 수 있다.

- IFS 값이 줄바꿈 문자만 인식할 수 있도록 변경

```
IFS=$'Wn'
```

## for 명령 (Cont.)

- 필드 구분자 변경하기
- 콜론으로 구분되는 파일의 값(예, /etc/passwd)을 차례대로 되풀이한다고 가정하면 아래와 같은 방법으로 설정해야 한다

```
IFS=:
```

- IFS 글자를 둘 이상 지정하려면 문자열로 써 주면 된다.

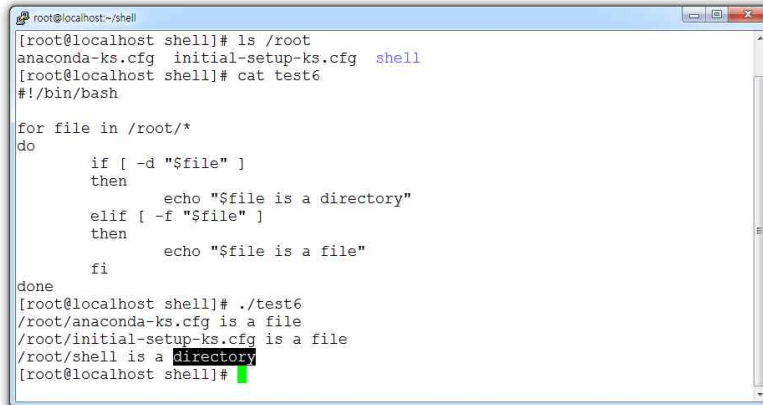
```
IFS=$'Wn':;'
```

여기서는 필드 구분 기호로 줄바꿈, 콜론, 세미콜론, 겹따옴표 문자를 사용한다. IFS로 어떤 문자를 써서 데이터를 분석할 수 있는가에 대한 제한은 없다.

## for 명령 (Cont.)

- 와일드카드를 써서 디렉토리 읽기

for 명령을 써서 어떤 디렉토리 안의 파일을 자동으로 차례차례 되풀이 할 수 있다. 이렇게 하려면 파일 또는 경로 이름에 와일드카드 문자를 사용한다. 그러면 셸은 강제로 파일 글로빙을 사용한다. 파일 글로빙은 지정된 와일드카드 문자와 일치 파일 이름 또는 경로 이름을 만들어 내는 과정이다



```

root@localhost:~/shell
[root@localhost shell]# ls /root
anaconda-ks.cfg  initial-setup-ks.cfg  shell
[root@localhost shell]# cat test6
#!/bin/bash

for file in /root/*
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    fi
done
[root@localhost shell]# ./test6
/root/anaconda-ks.cfg is a file
/root/initial-setup-ks.cfg is a file
/root/shell is a directory
[root@localhost shell]#
  
```

## for 명령 (Cont.)

- 와일드카드를 써서 디렉토리 읽기

여러 개의 디렉토리 와일드카드를 나열함으로써 같은 for 명령에서 디렉토리 검색방법과 나열 방법을 결합 할 수 도 있다.



```

root@localhost:~/shell
[root@localhost shell]# cat test7
#!/bin/bash

for file in /home/user1/.b* /home/user1/badtest
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    else
        echo "$file doesn't exist"
    fi
done
[root@localhost shell]# ./test7
/home/user1/.bash_logout is a file
/home/user1/.bash_profile is a file
/home/user1/.bashrc is a file
/home/user1/badtest doesn't exist
[root@localhost shell]#
  
```

## while 명령

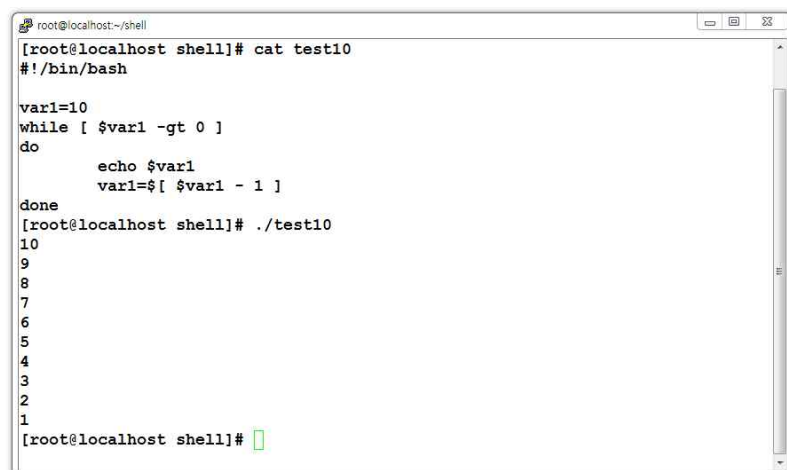
- while 명령은 테스트할 명령을 정의한 다음 테스트 명령이 종료 상태 0 을 되돌려 주는 동안에는 일련의 명령들을 되풀이 한다. while 명령은 반복이 시작될 때마다 테스트 명령을 실행하며, 테스트 명령이 0이 아닌 종료 상태를 되돌려 주면 구문 안에 있는 명령들의 반복을 중단한다.
- 기본 형식은 아래와 같다

```
while test command
do
    other commands
done
```

- while 명령에 정의된 테스트 명령은 if-then 구문과 동일하다. if-then 구문에서처럼 정상적인 bash 셸 명령이라면 뭐든 사용할 수 있으며 변수값과 같은 조건을 테스트 하기 위해 사용할 수도 있다

## while 명령 (Cont.)

- 가장 일반적인 사용은 루프 명령들이 사용하는 셸 변수의 값을 검사하기 위해 대괄호를 사용하는 것이다. (테스트 조건이 더 이상 true 가 아니라면 while 루프는 중단된다)



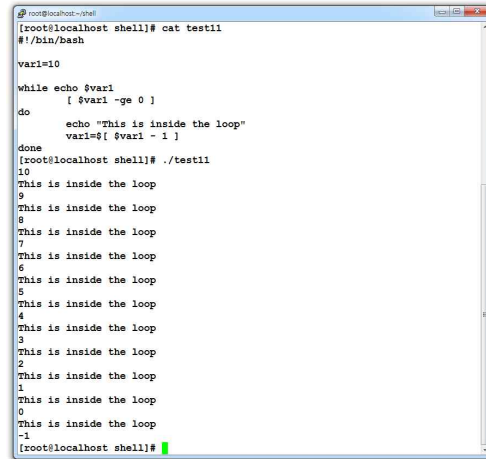
```
root@localhost:~/shell
[root@localhost shell]# cat test10
#!/bin/bash

var1=10
while [ $var1 -gt 0 ]
do
    echo $var1
    var1=$(( $var1 - 1 ])
done
[root@localhost shell]# ./test10
10
9
8
7
6
5
4
3
2
1
[root@localhost shell]#
```

## while 명령 (Cont.)

- 여러 테스트 명령 사용하기

while 문의 줄에 여러 테스트 명령을 정의할 수도 있다. 루프를 중단시킬 여부는 마지막 테스트 명령의 종료 상태만으로 결정된다. 아래의 예는 조심해야 할 부분을 설명한다



```
[root@localhost shell]# cat test11
#!/bin/bash

var1=10

while echo $var1
[ $var1 -ge 0 ]
do
    echo "This is inside the loop"
    var1=$(( $var1 - 1 ))
done
[root@localhost shell]# ./test11
10
This is inside the loop
9
This is inside the loop
8
This is inside the loop
7
This is inside the loop
6
This is inside the loop
5
This is inside the loop
4
This is inside the loop
3
This is inside the loop
2
This is inside the loop
1
This is inside the loop
0
This is inside the loop
-1
[root@localhost shell]#
```

## while 명령 (Cont.)

- 여러 테스트 명령 사용하기

```
while echo $var1
[ $var1 -ge 0 ]
```

첫번째 테스트는 단순히 var1의 변수를 표시하고 두번째 테스트는 var1 변수의 값을 판단하기 위해 대괄호를 사용한다. 루프 안에서 echo 문은 루프가 처리되었음을 뜻하는 단순한 메시지를 표시한다. 이 예제는 아래의 결과를 만들어 낸다

```
This is inside the loop
-1
```

while 루프는 var1 변수가 0이 되었을 때 echo 문을 실행시켰고 var1 변수의 값을 1감소시켰다. 그 다음, 테스트 명령이 다음 반복 때 실행되었다. echo 테스트 명령이 실행되었고, var1 변수의 값을 표시했다. 값은 0 보다 작다. 아직 shell이 while 루프를 끝낼 테스트 명령을 실행하지 않는다.

위 예제는 while 문에 다중 명령을 넣으면 각 반복 때마다, 마지막으로 되풀이할 때 실패하는 마지막 테스트 명령을 포함한 모든 명령을 실행하는 것을 보여준다. 이 점을 주의 해야 한다. 각 테스트 명령은 별개의 줄에 있어야 한다

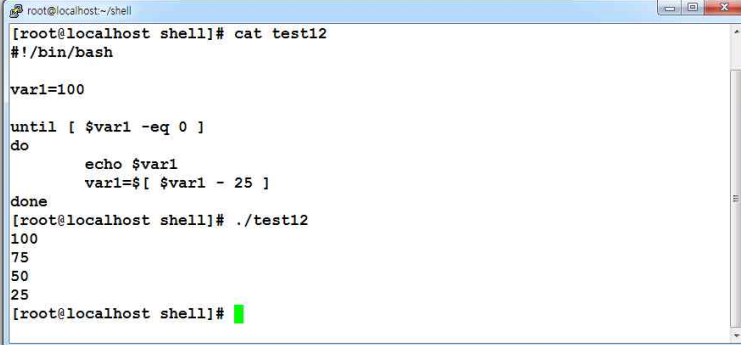
## Until 명령

- Until 명령은 while 와 반대로 동작한다  
until 명령에는 보통은 0이 아닌 종료 상태를 만들어내는 테스트 명령을 지정해야 한다. 테스트 명령의 종료 상태가 0이 아닌 한 bash 셸은 루프에 들어있는 명령들을 실행한다. 테스트 명령이 종료 상태 0을 돌려주면 루프는 중단된다.
- until 명령의 형식  

```
until test commands
do
    other commands
done
```
- while 명령들과 비슷하게 until 명령도 둘 이상의 테스트 명령을 가질 수 있다. 마지막 명령의 종료 상태만이 bash 셸이 정의된 다른 명령을 실행할 지 여부를 결정한다

## Until 명령 (Cont.)

- Until 명령의 사용 예



```

root@localhost:~/shell
[root@localhost shell]# cat test12
#!/bin/bash

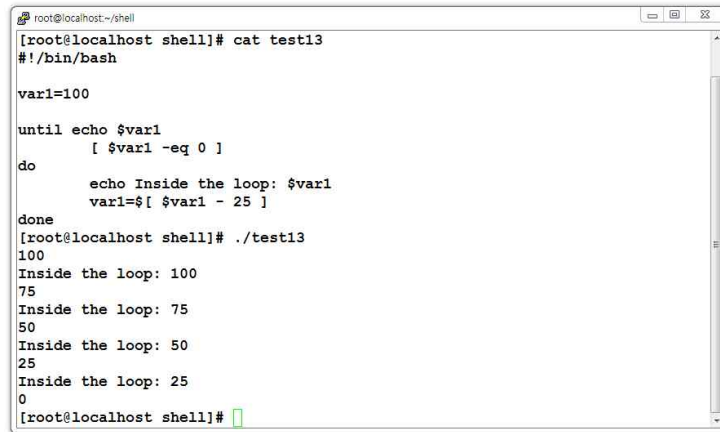
var1=100

until [ $var1 -eq 0 ]
do
    echo $var1
    var1=$(( $var1 - 25 ])
done
[root@localhost shell]# ./test12
100
75
50
25
[root@localhost shell]#
  
```

위의 예는 루프를 중단해야할 시점을 결정하기 위해 var1 변수를 테스트한다. 변수의 값이 0이 되는 즉시 명령 루프는 중단된다. 여러 테스트 명령을 사용할 때 while 명령에 적용되었던 주의사항이 until 에도 동일하게 적용된다

## Until 명령 (Cont.)

- Until 명령의 사용 예



```

root@localhost:~/shell
[root@localhost shell]# cat test13
#!/bin/bash

var1=100

until echo $var1
[ $var1 -eq 0 ]
do
    echo Inside the loop: $var1
    var1=$(( $var1 - 25 ))
done
[root@localhost shell]# ./test13
100
Inside the loop: 100
75
Inside the loop: 75
50
Inside the loop: 50
25
Inside the loop: 25
0
[root@localhost shell]#
  
```

셸은 지정된 테스트 명령을 실행하고 마지막 명령이 true 일 때만 중단한다

## Select 명령과 메뉴

- 루프의 한 종류로 사용자의 입력을 받아서 처리하게 된다. 이때 입력값은 메뉴 목록의 숫자 중 하나가 되어야 한다. 사용자의 입력을 받기 위해서는 PS3 프롬프트를 사용한다. 기본 형식은 다음과 같다. 사용자의 입력값은 REPLY 변수에 저장된다

```

select var in list
do
    command
done
  
```

```

[root@localhost shell]# cat select1
#!/bin/bash

PS3="Select the program you want to excute : "

select program in 'ls -l' pwd date exit
do
    $program
done
[root@localhost shell]# ./select1
1) ls -l
2) pwd
3) date
4) exit
Select the program you want to excute :2
/root/shell
Select the program you want to excute :3
Thu Feb  6 08:22:40 KST 2020
Select the program you want to excute :4
[root@localhost shell]#
  
```

## Select 명령과 메뉴 (Cont.)

- case 명령은 메뉴와 선택, 명령 실행으로부터 유저가 선택하도록 하는 select 명령과 함께 사용될 수 있다.

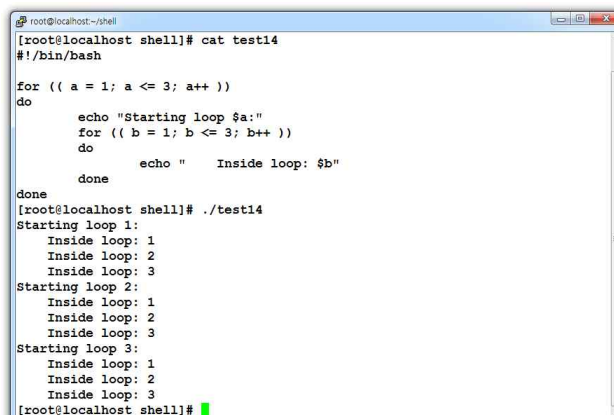
```
[root@localhost shell]# cat select2
#!/bin/bash

PS3="웹 사이트 주소를 볼 수 있습니다:"
select choice in 구글 구글코리아 네이버 다음 종료
do
    case "$choice" in
        구글 | 구글코리아 )
            echo "구글 - http://www.google.com"
            continue;;
        네이버 )
            echo "네이버 - http://www.naver.com"
            continue;;
        다음 )
            echo "다음 - http://www.daum.net"
            continue;;
        종료 )
            echo "finished"
            break;;
        *)
            echo "$REPLY는 없습니다. 1~4만 선택하세요"
            echo "다시 선택"
            ;;
    esac
done
[root@localhost shell]#
```

```
[root@localhost shell]# ./select2
1) 구글
2) 구글코리아
3) 네이버
4) 다음
5) 종료
웹 사이트 주소를 볼 수 있습니다:1
구글 - http://www.google.com
웹 사이트 주소를 볼 수 있습니다:3
네이버 - http://www.naver.com
웹 사이트 주소를 볼 수 있습니다:5
finished
[root@localhost shell]# ./select2
1) 구글
2) 구글코리아
3) 네이버
4) 다음
5) 종료
웹 사이트 주소를 볼 수 있습니다:7
7는 없습니다. 1~4만 선택하세요
다시 선택
웹 사이트 주소를 볼 수 있습니다:5
finished
[root@localhost shell]#
```

## 중첩된 루프

- 루프 문은 루프 안에 어떤 종류의 명령이든 포함할 수 있으며 여기에는 다른 루프 명령도 포함된다. 이를 중첩 루프라고 한다. 중첩 루프를 사용할 때에는 주의가 필요하다. 반복 과정 안에서 또 다른 반복이 이루어질 때 실행되는 명령의 횟수는 곱으로 늘어난다. 이에 주의하지 않으면 스크립트에 문제가 일어날 수 있다
- 아래는 for 루프 안에 중첩된 for 루프가 들어 있는 예다.



```

[root@localhost ~]# cat test14
#!/bin/bash

for (( a = 1; a <= 3; a++ ))
do
    echo "Starting loop $a:"
    for (( b = 1; b <= 3; b++ ))
    do
        echo "    Inside loop: $b"
    done
done

[root@localhost shell]# ./test14
Starting loop 1:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3
Starting loop 2:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3
Starting loop 3:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3
[root@localhost shell]#
  
```



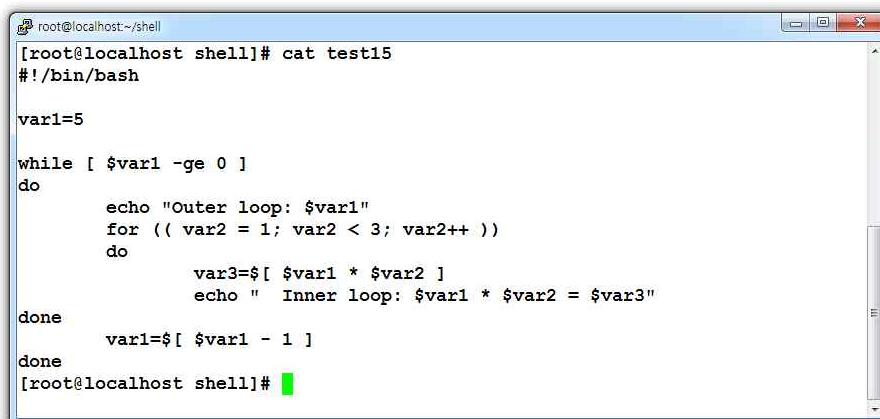
## 중첩된 루프 (Cont.)

- 앞의 예에 대한 설명

바깥쪽 루프가 반복될 때마다 안쪽 루프가 반복된다. 두 루프의 do 와 done 명령 사이에 있는 명령들은 차이가 없다는 것을 알 수 있다. bash 셸은 첫 번째 done 명령이 수행 될 때 이것이 바깥쪽 루프가 아닌 안쪽 루프의 것임을 안다

## 중첩된 루프 (Cont.)

- while 루프 안에 for 문을 놓을 때와 같이 여러 종류의 루프 명령을 혼합할 때에도 이러한 규칙이 적용된다



```

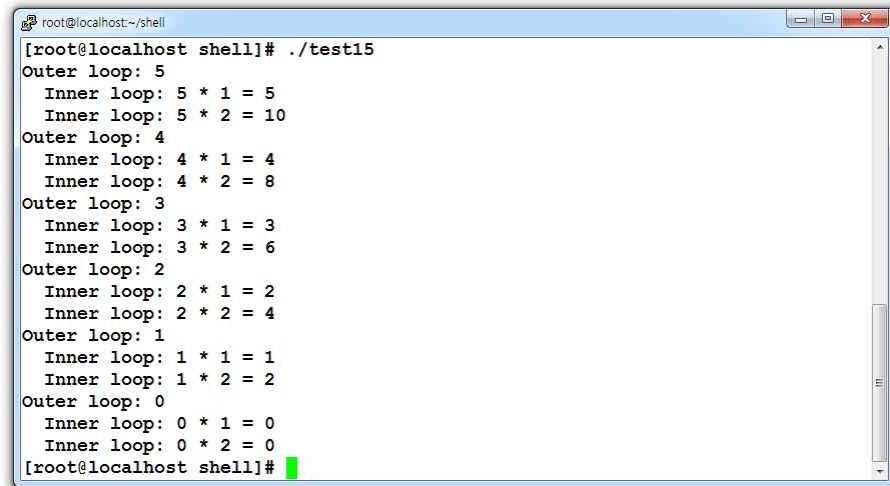
root@localhost:~/shell
[root@localhost shell]# cat test15
#!/bin/bash

var1=5

while [ $var1 -ge 0 ]
do
    echo "Outer loop: $var1"
    for (( var2 = 1; var2 < 3; var2++ ))
    do
        var3=$(( $var1 * $var2 ))
        echo "  Inner loop: $var1 * $var2 = $var3"
    done
    var1=$(( $var1 - 1 ))
done
[root@localhost shell]#
  
```

## 중첩된 루프 (Cont.)

- 앞의 예에 대한 결과



```

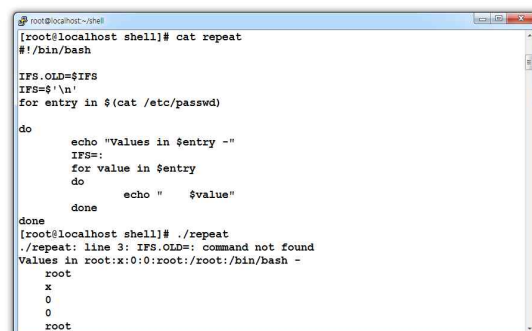
root@localhost:~/shell
[root@localhost shell]# ./test15
Outer loop: 5
  Inner loop: 5 * 1 = 5
  Inner loop: 5 * 2 = 10
Outer loop: 4
  Inner loop: 4 * 1 = 4
  Inner loop: 4 * 2 = 8
Outer loop: 3
  Inner loop: 3 * 1 = 3
  Inner loop: 3 * 2 = 6
Outer loop: 2
  Inner loop: 2 * 1 = 2
  Inner loop: 2 * 2 = 4
Outer loop: 1
  Inner loop: 1 * 1 = 1
  Inner loop: 1 * 2 = 2
Outer loop: 0
  Inner loop: 0 * 1 = 0
  Inner loop: 0 * 2 = 0
[root@localhost shell]#
  
```

## 파일 데이터에 대한 반복 작업

- 파일 내부에 저장된 항목에 대해 차례대로 반복작업을 해야 할 때가 있다. 이를 위해서는 앞에서 다뤘던 두 가지 기술을 결합해야 한다

- 중첩 루프 사용
- IFS 환경 변수 변경

- 다음은 위의 두가지 기술을 혼합한 예이다



```

root@localhost:~/shell
[root@localhost shell]# cat repeat
#!/bin/bash

IFS_OLD=$IFS
IFS=$'\n'
for entry in $(cat /etc/passwd)
do
    echo "Values in $entry -"
    IFS=:
    for value in $entry
    do
        echo "    $value"
    done
done
[root@localhost shell]# ./repeat
./repeat: line 3: IFS_OLD=: command not found
Values in root:x:0:0:root:/root:/bin/bash -
    root
    x
    0
    0
    root
  
```

## 루프제어

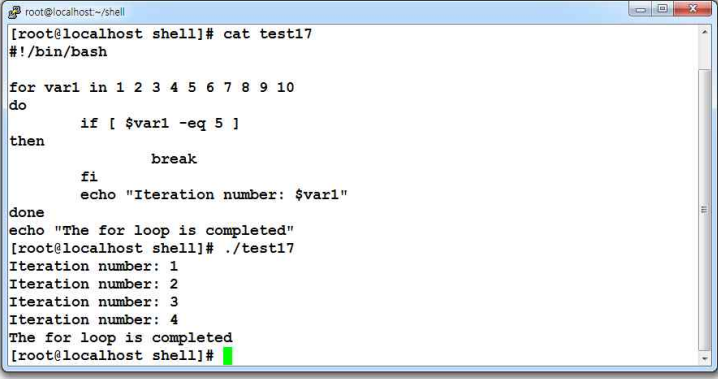
- 루프를 시작한 이후 루프의 모든 반복 작업을 완료할 때까지 루프 안쪽의 일들을 제어하는 데 다음의 명령을 사용할 수 있다
  - break 명령
  - continue 명령
- break 명령  
break 명령은 진행 중인 루프에서 탈출 할 수 있는 간단한 방법이다. while 와 until 루프까지 포함하여 어떤 유형의 루프든 break 명령으로 빠져나올 수 있다.

여러 가지 상황에서 break 명령을 사용할 수 있다

## 루프제어 (Cont.)

- break 명령

단일 루프 빠져나오기



```

root@localhost:~/shell
[root@localhost shell]# cat test17
#!/bin/bash

for var1 in 1 2 3 4 5 6 7 8 9 10
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Iteration number: $var1"
done
echo "The for loop is completed"
[root@localhost shell]# ./test17
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
The for loop is completed
[root@localhost shell]#
  
```

루프는 일반적으로 목록에 지정된 모든 값에 걸쳐 명령을 반복해야 한다. 그러나 if-then 조건이 충족되었을 때 셸은 루프를 중지시키는 break 명령을 실행한다

## 루프제어 (Cont.)

- break 명령

안쪽 루프 빠져나오기

```
[root@localhost shell]# cat test19
#!/bin/bash

for (( a = 1; a < 4; a++ ))
do
    echo "Outer loop: $a"
    for (( b=1; b < 100; b++ ))
    do
        if [ $b -eq 5 ]
        then
            break
        fi
        echo "    Inner loop: $b"
    done
done
[root@localhost shell]#
```

```
[root@localhost shell]# ./test19
Outer loop: 1
    Inner loop: 1
    Inner loop: 2
    Inner loop: 3
    Inner loop: 4
Outer loop: 2
    Inner loop: 1
    Inner loop: 2
    Inner loop: 3
    Inner loop: 4
Outer loop: 3
    Inner loop: 1
    Inner loop: 2
    Inner loop: 3
    Inner loop: 4
[root@localhost shell]#
[root@localhost shell]#
```

for 문은 b 변수가 100이 될 때까지 되풀이 되도록 지정되어 있다. 그러나 내부 루프의 if-then 구문은 변수 b의 값이 5와 같으면 break 명령이 실행되도록 지정되어 있다

## 루프제어 (Cont.)

- break 명령

바깥쪽 루프 빠져나오기

```
[root@localhost shell]# cat test20
#!/bin/bash

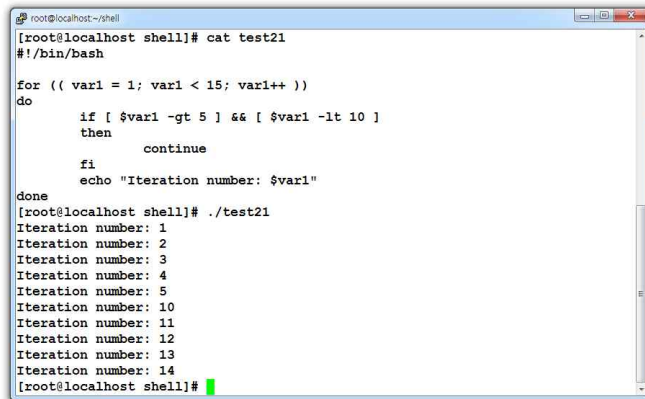
for (( a = 1; a < 4; a++ ))
do
    echo "Outer loop: $a"
    for (( b = 1; b < 100; b++ ))
    do
        if [ $b -gt 4 ]
        then
            break 2
        fi
        echo "    Inner loop: $b"
    done
done
[root@localhost shell]# ./test20
Outer loop: 1
    Inner loop: 1
    Inner loop: 2
    Inner loop: 3
    Inner loop: 4
[root@localhost shell]#
```

헬이 break 명령을 실행했을 때 바깥쪽 루프도 중지된다

## 루프제어 (Cont.)

- continue 명령

continue 명령은 루프의 내부 명령의 처리를 일찍 중지키시지만 루프를 완전히 종료하지는 않도록 하는 방법이다. 이 명령은 루프 안에서 쉘이 명령을 실행하지 않도록 하는 조건을 설정할 수 있다. 다음을 for 루프에서 continue 명령을 사용하는 간단한 예이다



```

root@localhost:~/shell
[root@localhost shell]# cat test21
#!/bin/bash
for (( var1 = 1; var1 < 15; var1++ ))
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
        continue
    fi
    echo "Iteration number: $var1"
done
[root@localhost shell]# ./test21
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
Iteration number: 10
Iteration number: 11
Iteration number: 12
Iteration number: 13
Iteration number: 14
[root@localhost shell]#
  
```

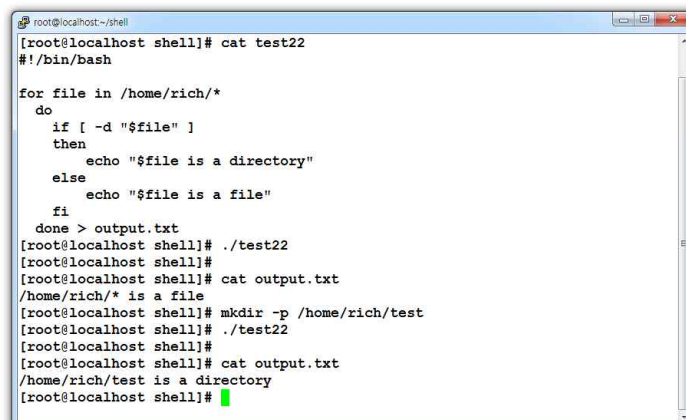
if-then 구문의 조건이 충족될 때 (5보다 크고 10보다 작을 때) 쉘은 continue 명령을 실행시켜서 루프에서 명령의 나머지 부분을 건너뛰지만 루프 자체는 유지

if-then 조건이 더 이상 충족되지 않으면 상황은 정상으로 돌아온다

## 루프제어 (Cont.)

- 루프의 출력 처리하기

셸 스크립트에서 루프의 출력을 파이프 또는 리다이렉트할 수 있다. done 명령의 끝에 이러한 처리 명령을 덧붙이면 된다



```

root@localhost:~/shell
[root@localhost shell]# cat test22
#!/bin/bash

for file in /home/rich/*
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    else
        echo "$file is a file"
    fi
done > output.txt
[root@localhost shell]# ./test22
[root@localhost shell]# cat output.txt
/home/rich/* is a file
[root@localhost shell]# mkdir -p /home/rich/test
[root@localhost shell]# ./test22
[root@localhost shell]# cat output.txt
/home/rich/test is a directory
[root@localhost shell]#
  
```

## 사용자 입력 처리

- 교육내용
  - 매개변수 전달하기
  - 매개변수 추적하기
  - 시프트 기능 활용하기
  - 옵션 다루기
  - 옵션 표준화하기
  - 사용자 입력 얻기

## 매개변수 전달하기

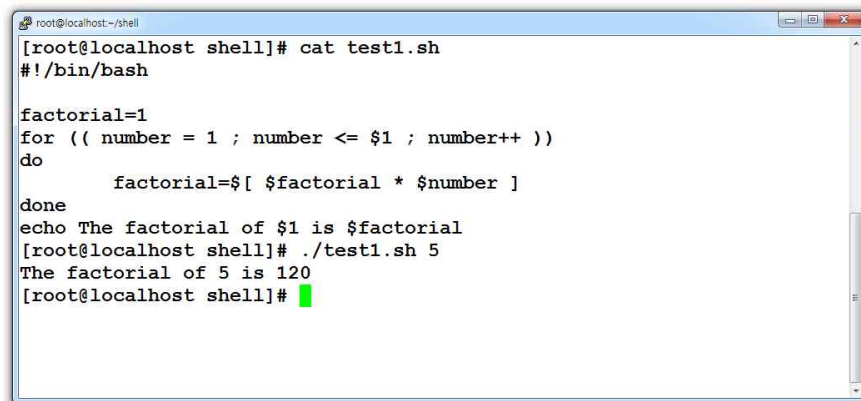
- 셸 스크립트에 데이터를 전달하는 가장 기본적인 방법은 커맨드라인 매개변수를 사용하는 것이다 커맨드라인 매개변수는 스크립트를 실행할 때 커맨드라인에 데이터를 추가 할 수 있다

```
# ./addem 10 30
```

위의 예는 스크립트 addem 에 두 개의 커맨드라인 매개변수 10과 30을 전달한다. 스크립트는 특수한 변수를 사용하여 커맨드라인 매개변수를 다룬다

## 매개변수 읽기

- bash 셸은 입력된 커맨드라인 매개변수를 위치 매개변수라고 하는 특별한 변수에 할당한다. 여기에는 셸이 실행되는 스크립트의 이름이 포함된다. 위치 매개변수는 보통의 숫자로 \$0이 스크립트의 이름, \$1이 첫번째 매개변수, \$2는 두 번째 매개변수와 같은 식으로 \$9까지 이어진다



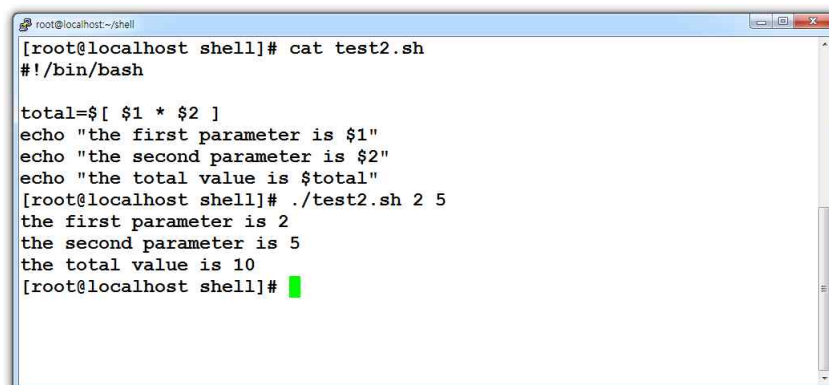
```

root@localhost:~/shell
[root@localhost shell]# cat test1.sh
#!/bin/bash

factorial=1
for (( number = 1 ; number <= $1 ; number++ ))
do
    factorial=$(( factorial * $number ))
done
echo The factorial of $1 is $factorial
[root@localhost shell]# ./test1.sh 5
The factorial of 5 is 120
[root@localhost shell]#
  
```

## 매개변수 읽기 (Cont.)

- 더 많은 커맨드라인 매개변수를 입력시 각 매개변수를 커맨드라인에서 빈칸으로 구분한다



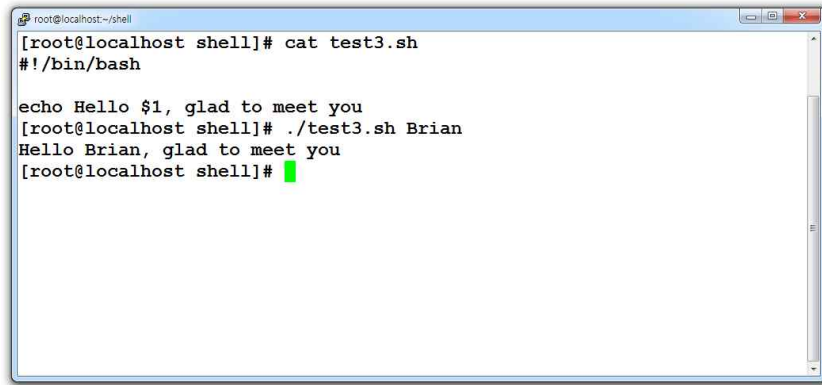
```

root@localhost:~/shell
[root@localhost shell]# cat test2.sh
#!/bin/bash

total=$(( $1 * $2 ))
echo "the first parameter is $1"
echo "the second parameter is $2"
echo "the total value is $total"
[root@localhost shell]# ./test2.sh 2 5
the first parameter is 2
the second parameter is 5
the total value is 10
[root@localhost shell]#
  
```

## 매개변수 읽기 (Cont.)

- 커맨드 라인에서 텍스트 문자열 사용하기



```

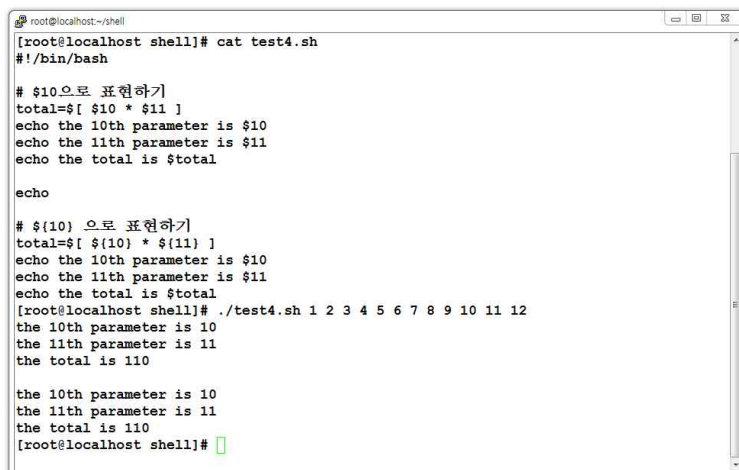
root@localhost:~/shell
[root@localhost shell]# cat test3.sh
#!/bin/bash

echo Hello $1, glad to meet you
[root@localhost shell]# ./test3.sh Brian
Hello Brian, glad to meet you
[root@localhost shell]#
  
```

- 매개변수 값에 빈 칸을 포함하려면 따옴표( ' 또는 " ) 를 사용해야 한다

## 매개변수 읽기 (Cont.)

- 9개 이상의 매개변수 사용시 10번째 부터는 \${10}, \${10} 과 같이 표현한다



```

root@localhost:~/shell
[root@localhost shell]# cat test4.sh
#!/bin/bash

# $10으로 표현하기
total=$(( $10 * $11 ))
echo the 10th parameter is $10
echo the 11th parameter is $11
echo the total is $total

echo

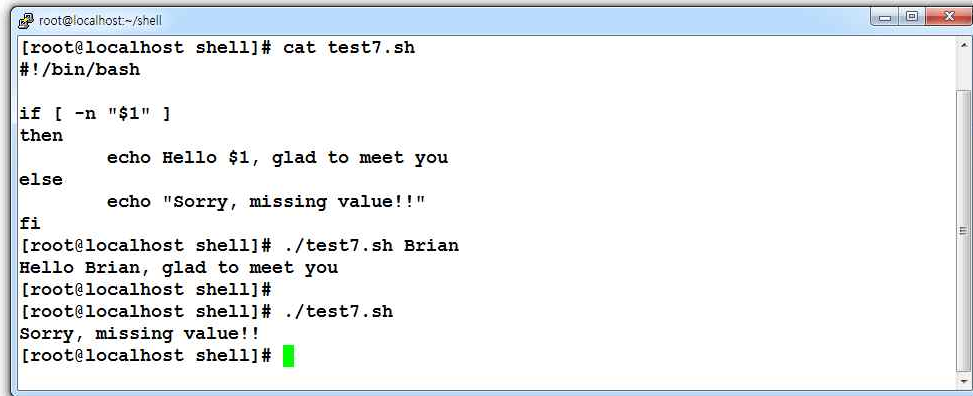
# ${10} 으로 표현하기
total=$(( ${10} * ${11} ))
echo the 10th parameter is $10
echo the 11th parameter is $11
echo the total is $total
[root@localhost shell]# ./test4.sh 1 2 3 4 5 6 7 8 9 10 11 12
the 10th parameter is 10
the 11th parameter is 11
the total is 110

the 10th parameter is 10
the 11th parameter is 11
the total is 110
[root@localhost shell]#
  
```



## 매개변수 읽기 (Cont.)

- 매개변수가 없을 경우 'syntax error' 를 발생시킨다. 매개변수는 데이터가 있다고 가정하므로 데이터가 존재하지 않을 때에는 스크립트에서 오류 메시지가 나타날 가능성이 높다. 아래와 같은 예를 사용해 본다



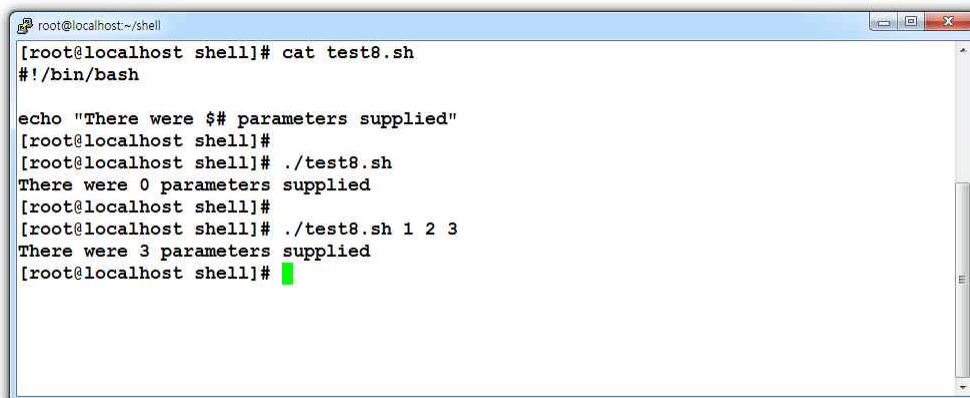
```

root@localhost:~/shell
[root@localhost shell]# cat test7.sh
#!/bin/bash

if [ -n "$1" ]
then
    echo Hello $1, glad to meet you
else
    echo "Sorry, missing value!!"
fi
[root@localhost shell]# ./test7.sh Brian
Hello Brian, glad to meet you
[root@localhost shell]#
[root@localhost shell]# ./test7.sh
Sorry, missing value!!
[root@localhost shell]#
  
```

## 특수한 매개변수

- \$# 는 스크립트를 실행할 때의 커맨드라인 매개변수의 수가 포함되어 있다. 일반 변수처럼 스크립트의 아무 곳에서나 이 특수 변수를 사용할 수 있다



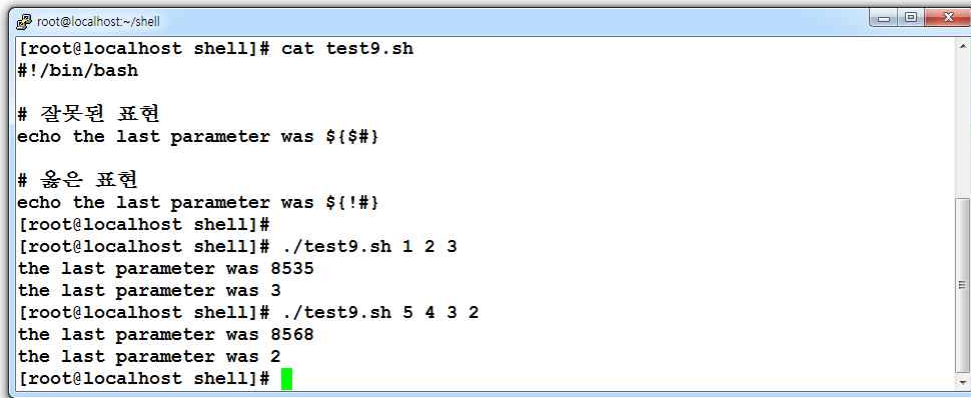
```

root@localhost:~/shell
[root@localhost shell]# cat test8.sh
#!/bin/bash

echo "There were $# parameters supplied"
[root@localhost shell]#
[root@localhost shell]# ./test8.sh
There were 0 parameters supplied
[root@localhost shell]#
[root@localhost shell]# ./test8.sh 1 2 3
There were 3 parameters supplied
[root@localhost shell]#
  
```

## 특수한 매개변수 (Cont.)

- 마지막 매개변수를 알고자 할 때에는 \$# 를 중괄호 안에서 사용할 수 없어 느낌표로 변경해 주어야 정상동작한다



```

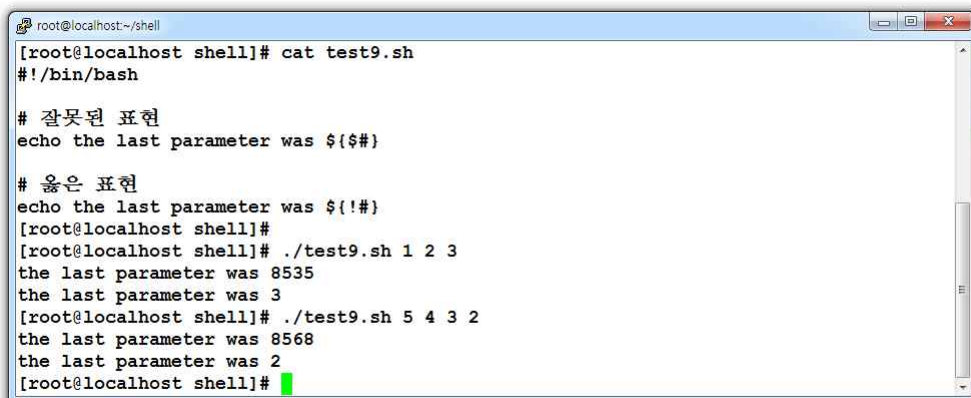
root@localhost:~/shell
[root@localhost shell]# cat test9.sh
#!/bin/bash

# 잘못된 표현
echo the last parameter was ${${#}}

# 옳은 표현
echo the last parameter was ${!#}
[root@localhost shell]#
[root@localhost shell]# ./test9.sh 1 2 3
the last parameter was 8535
the last parameter was 3
[root@localhost shell]# ./test9.sh 5 4 3 2
the last parameter was 8568
the last parameter was 2
[root@localhost shell]#
  
```

## 특수한 매개변수 (Cont.)

- 마지막 매개변수를 알고자 할 때에는 \$# 를 중괄호 안에서 사용할 수 없어 느낌표로 변경해 주어야 정상동작한다



```

root@localhost:~/shell
[root@localhost shell]# cat test9.sh
#!/bin/bash

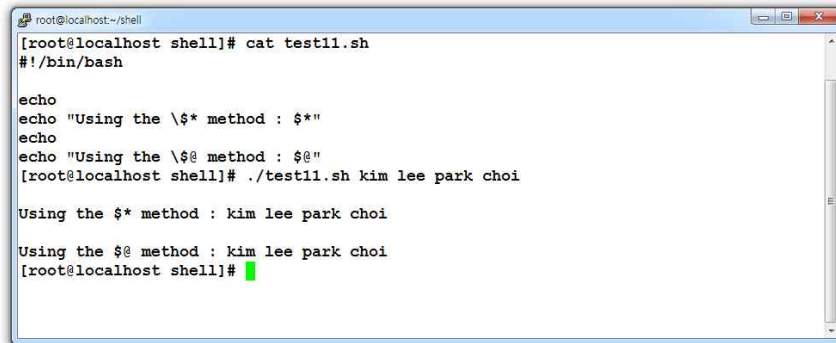
# 잘못된 표현
echo the last parameter was ${${#}}

# 옳은 표현
echo the last parameter was ${!#}
[root@localhost shell]#
[root@localhost shell]# ./test9.sh 1 2 3
the last parameter was 8535
the last parameter was 3
[root@localhost shell]# ./test9.sh 5 4 3 2
the last parameter was 8568
the last parameter was 2
[root@localhost shell]#
  
```

## 특수한 매개변수 (Cont.)

- 모든 데이터를 한꺼번에 얻기

커맨드라인에 제공되는 모든 매개변수를 한꺼번에 얻어야 할 때도 있다. 얼마나 많은 매개변수가 커맨드라인에 있는지 판단하고 이들 모두를 순서대로 되풀이하기 위해서 \$# 변수를 사용하는 대신 \$\* 과 @\$ 변수를 사용할 수 있다



```

root@localhost:~/shell
[root@localhost shell]# cat test11.sh
#!/bin/bash

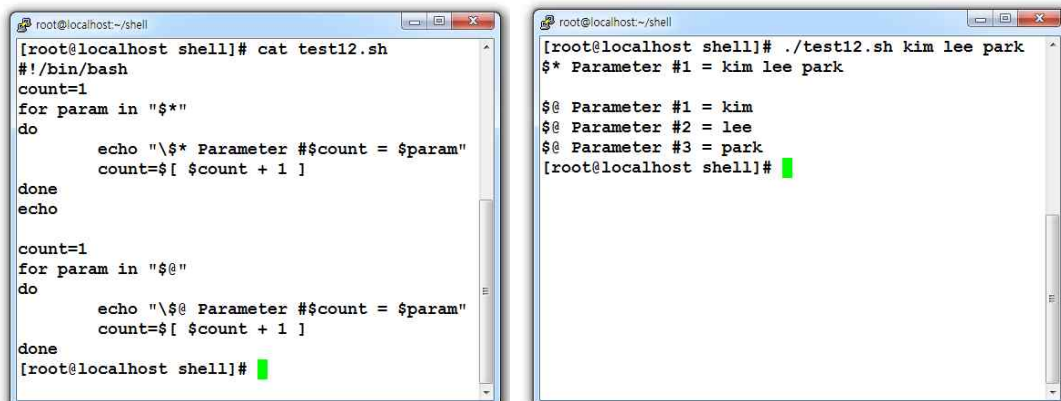
echo
echo "Using the \$* method : $*"
echo
echo "Using the @$ method : @$"
[root@localhost shell]# ./test11.sh kim lee park choi

Using the $* method : kim lee park choi

Using the @$ method : kim lee park choi
[root@localhost shell]#
  
```

## 특수한 매개변수 (Cont.)

- \$\* 과 @\$ 의 차이점



```

root@localhost:~/shell
[root@localhost shell]# cat test12.sh
#!/bin/bash
count=1
for param in "$*"
do
    echo "\$* Parameter #$count = $param"
    count=$((count + 1))
done
echo

count=1
for param in "$@"
do
    echo "\@$@ Parameter #$count = $param"
    count=$((count + 1))
done
[root@localhost shell]#

root@localhost:~/shell
[root@localhost shell]# ./test12.sh kim lee park
$* Parameter #1 = kim lee park

$@ Parameter #1 = kim
$@ Parameter #2 = lee
$@ Parameter #3 = park
[root@localhost shell]#
  
```

\$\*은 모두 매개변수를 하나로 다루고 @\$는 각각의 매개변수를 분리해서 다룬다

## 사용자 입력받기

- 커맨드라인 옵션 및 매개변수는 스크립트가 사용자의 데이터를 얻을 수 있는 좋은 방법이지만 때로는 스크립트는 좀 더 상호작용이 필요하다. 스크립트는 실행되는 동안 질문을 하고 스크립트를 실행하는 사람의 응답을 기다려야 한다. bash 셸은 이 목적을 위해 read 명령을 제공한다



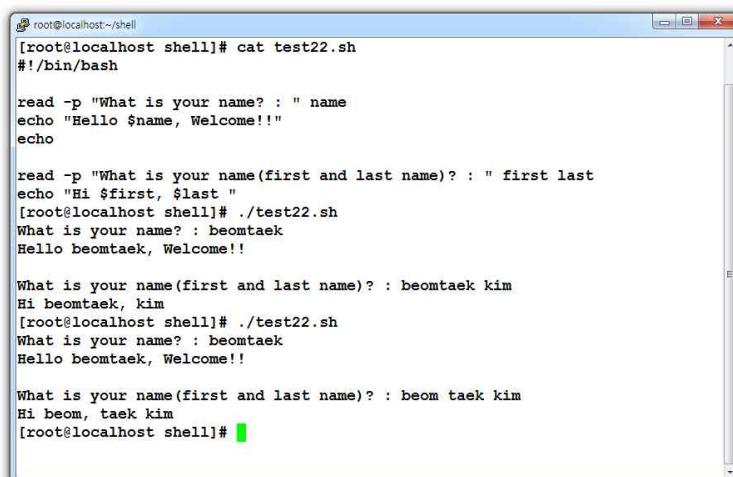
```

root@localhost:~/shell
[root@localhost shell]# cat test21.sh
#!/bin/bash

echo -n "Enter your name: "
read name
echo "Hello $name, welcome!"
[root@localhost shell]# ./test21.sh
Enter your name: beomtaek
Hello beomtaek, welcome!
[root@localhost shell]#
  
```

## 사용자 입력받기 (Cont.)

- read 커맨드라인에 직접 메시지를 지정할 수 있는 -p 옵션을 포함한다



```

root@localhost:~/shell
[root@localhost shell]# cat test22.sh
#!/bin/bash

read -p "What is your name? : " name
echo "Hello $name, Welcome!!"
echo

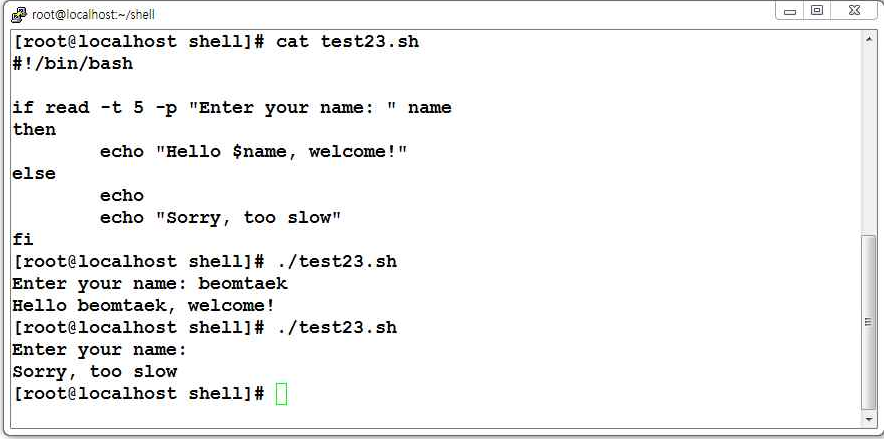
read -p "What is your name(first and last name)? : " first last
echo "Hi $first, $last "
[root@localhost shell]# ./test22.sh
What is your name? : beomtaek
Hello beomtaek, Welcome!!

What is your name(first and last name)? : beomtaek kim
Hi beomtaek, kim
[root@localhost shell]# ./test22.sh
What is your name? : beomtaek
Hello beomtaek, Welcome!!

What is your name(first and last name)? : beom taek kim
Hi beom, taek kim
[root@localhost shell]#
  
```

## 사용자 입력받기 (Cont.)

- 시간초과

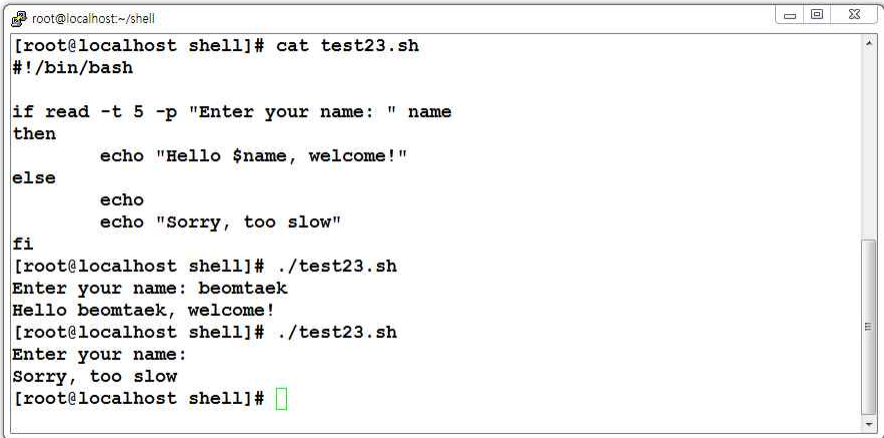


```
root@localhost:~/shell
[root@localhost shell]# cat test23.sh
#!/bin/bash

if read -t 5 -p "Enter your name: " name
then
    echo "Hello $name, welcome!"
else
    echo
    echo "Sorry, too slow"
fi
[root@localhost shell]# ./test23.sh
Enter your name: beomtaek
Hello beomtaek, welcome!
[root@localhost shell]# ./test23.sh
Enter your name:
Sorry, too slow
[root@localhost shell]#
```

## 사용자 입력받기 (Cont.)

- 시간초과

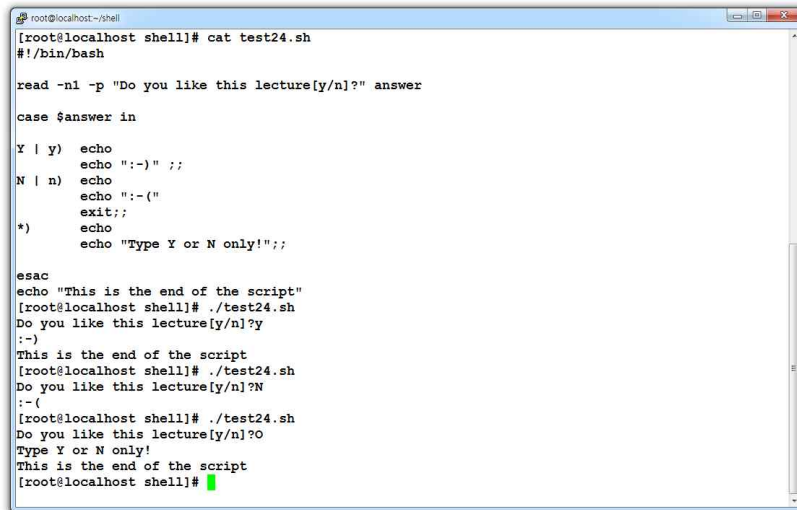


```
root@localhost:~/shell
[root@localhost shell]# cat test23.sh
#!/bin/bash

if read -t 5 -p "Enter your name: " name
then
    echo "Hello $name, welcome!"
else
    echo
    echo "Sorry, too slow"
fi
[root@localhost shell]# ./test23.sh
Enter your name: beomtaek
Hello beomtaek, welcome!
[root@localhost shell]# ./test23.sh
Enter your name:
Sorry, too slow
[root@localhost shell]#
```

## 사용자 입력받기 (Cont.)

- read 의 입력 글자수 제한하기



```

root@localhost:~/shell
[root@localhost shell]# cat test24.sh
#!/bin/bash

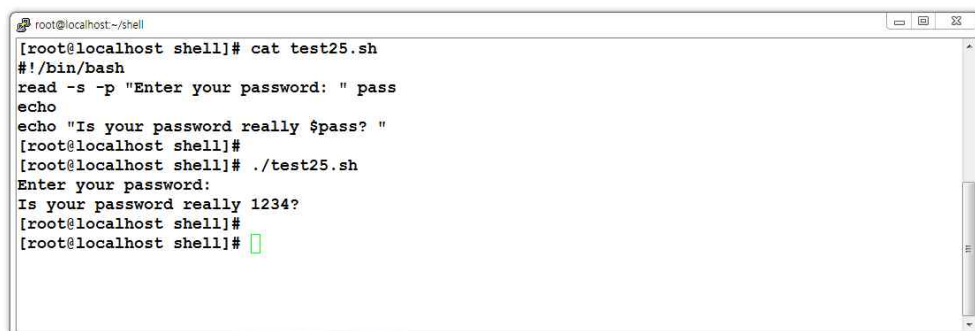
read -n1 -p "Do you like this lecture[y/n]?" answer

case $answer in
Y | y) echo
        echo ":-)" ;;
N | n) echo
        echo ":-("
        exit;;
*)
    echo "Type Y or N only!";;
esac

echo "This is the end of the script"
[root@localhost shell]# ./test24.sh
Do you like this lecture[y/n]?y
:-)
This is the end of the script
[root@localhost shell]# ./test24.sh
Do you like this lecture[y/n]?N
:-(
[root@localhost shell]# ./test24.sh
Do you like this lecture[y/n]?o
Type Y or N only!
This is the end of the script
[root@localhost shell]#
  
```

## 사용자 입력받기 (Cont.)

- -s 옵션은 read 명령이 입력 데이터를 모니터에 표시하지 못하도록 막는다. 실제로는 표시되지만 텍스트의 색이 배경색과 같게 설정되는 것이다



```

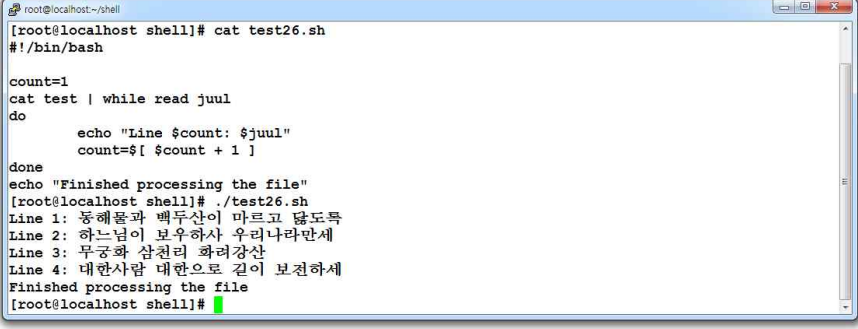
root@localhost:~/shell
[root@localhost shell]# cat test25.sh
#!/bin/bash
read -s -p "Enter your password: " pass
echo
echo "Is your password really $pass? "
[root@localhost shell]#
[root@localhost shell]# ./test25.sh
Enter your password:
Is your password really 1234?
[root@localhost shell]#
[root@localhost shell]#
  
```

## 사용자 입력받기 (Cont.)

- 파일에서 읽기

리눅스 시스템에서 파일에 저장된 데이터를 읽어 들일 때에도 read 명령을 사용할 수 있다. read 명령을 한 번 부를때마다 파일에서 텍스트 한 줄을 읽어 들인다. 더 이상 파일에 읽을 줄이 없으면 read 는 0 이 아닌 종료 상태와 함께 종료된다.

read 명령이 파일에서 데이터를 읽어 들이는 부분은 까다롭다. 보통 파일의 cat 명령의 결과를 파이프를 통해 곧바로 read 명령을 가지고 있는 whiel 명령에 전달한다.



```

root@localhost:~/shell
[root@localhost shell]# cat test26.sh
#!/bin/bash

count=1
cat test | while read juul
do
    echo "Line $count: $juul"
    count=$((count + 1))
done
echo "Finished processing the file"
[root@localhost shell]# ./test26.sh
Line 1: 동해물과 백두산이 마르고 닳도록
Line 2: 하늘님이 보우하사 우리나라만세
Line 3: 무궁화 삼천리 화려강산
Line 4: 대한사람 대한으로 길이 보전하세
Finished processing the file
[root@localhost shell]#
  
```

## 리눅스 셸 프로그래밍

### 응용부분

## gawk

- 교육내용
  - 기본 gawk
  - 정규표현식
  - 구조적 명령과 gawk

## 기본 gawk

- gawk 와 awk
  - gawk 는 유닉스에 있던 awk 의 GNU 버전
  - awk - 소규모 데이터베이스 관리, 보고서 생성, 색인 생성, 유효성 검사, 문서작업
  - gawk - awk 의 모든 기능을 포함하며, 데이터 정렬, 처리를 위해 비트 및 조각 데이터 추출이 용이하고 간단한 네트워크 통신을 수행하는데 도움이 되는 추가 기능 가능
  - 데이터를 저장하는 변수 정의
  - 데이터를 다룰 수 있도록 산술 및 문자열 연산자 사용
  - if-then 및 루프문과 같이 데이터 처리에 로직을 추가하는 구조적 프로그래밍 개념 사용
  - 데이터 파일 안에서 데이터 요소를 추출하고 다른 순서 또는 형식으로 재구성하여 형식화된 보고서 생성
  - 기본 형식

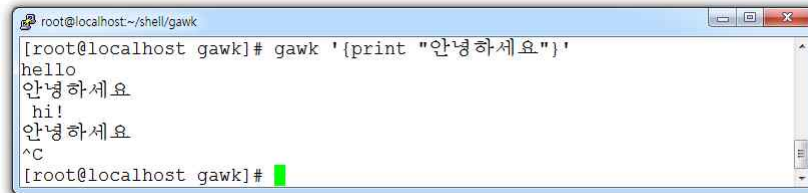
# gawk options program file

옵션	설명
-F fs	한 줄에서 데이터 필드의 경계를 식별하기 위한 파일 구분자를 지정한다
-f file	프로그램이 읽어 들일 파일 이름을 지정한다
-var =value	gawk 프로그램에서 사용할 변수의 기본값을 정의한다
-mf N	데이터 필드에서 처리할 필드의 최대 수를 지정한다
-mr N	데이터 파일의 최대 레코드 크기를 지정한다
-W keyword	gawk의 호환성 모드 또는 경고 수준을 지정한다



## 기본 gawk (Cont.)

- gawk 의 실행



```

root@localhost:~/shell/gawk
[root@localhost gawk]# gawk '{print "안녕하세요"}'
hello
안녕하세요
hi!
안녕하세요
^C
[root@localhost gawk]#
  
```

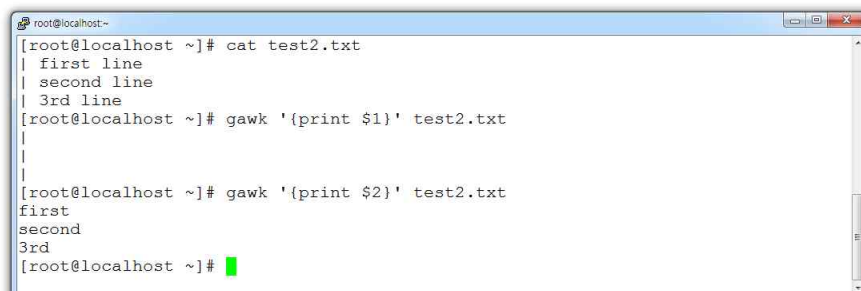
위의 예는 한 개의 명령인 print 명령을 정의한 것이다. 위 명령을 실행하면 아무런 결과도 나타나지 않는다. 커맨드라인에서 파일의 이름을 정의하지 않았기 때문에 gawk 는 표준입력에서 데이터를 검색한다. 즉, 표준입력으로 들어오는 텍스트를 기다리는 것이다. 텍스트를 입력하고 enter 키를 누르면 gawk는 스크립트를 통해 텍스트를 처리한다

위의 예는 고정된 텍스트 스트링을 표시하도록 설정되어 있으므로 데이터 스트림에 어떠한 데이터를 입력해도 텍스트 출력으로 얻는 것은 동일하다

## 기본 gawk (Cont.)

- 데이터 필드 변수 사용하기

\$0 : 텍스트의 전체 줄  
 \$1 : 텍스트의 줄에서 첫 번째 데이터 필드  
 \$2 : 텍스트의 줄에서 두 번째 데이터 필드



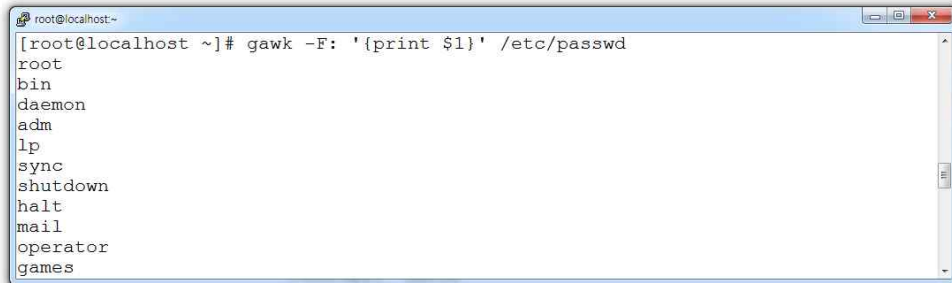
```

root@localhost:~
[root@localhost ~]# cat test2.txt
| first line
| second line
| 3rd line
[root@localhost ~]# gawk '{print $1}' test2.txt
|
|
|
[root@localhost ~]# gawk '{print $2}' test2.txt
first
second
3rd
[root@localhost ~]#
  
```

## 기본 gawk (Cont.)

- 데이터 필드 변수 사용하기

다른 필드 구분자를 사용하는 파일을 읽는다면 -F 옵션을 사용하여 구분자를 지정할 수 있다



```

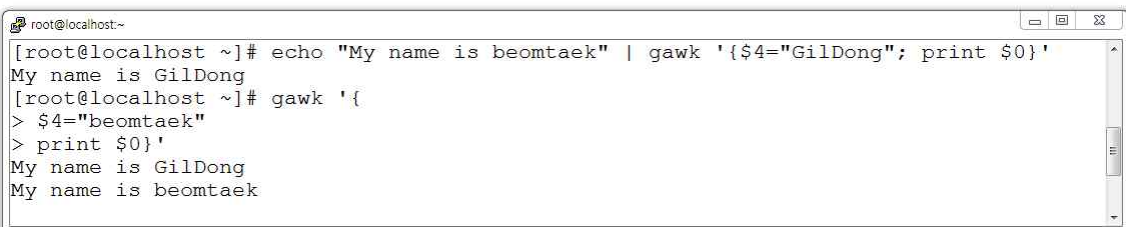
root@localhost:~# gawk -F: '{print $1}' /etc/passwd
root
bin
daemon
adm
lp
sync
shutdown
halt
mail
operator
games
  
```

Q. 너무 많은 라인을 출력했다. 위에서 10줄의 결과만 나오도록 하라

## 기본 gawk (Cont.)

- 프로그램 스크립트에서 여러 명령 사용하기

gawk 는 여러 가지 명령어를 프로그램에 넣을 수 있다. 커맨드라인에 지정된 프로그램 스크립트에서 여러 명령을 사용하려면 각 명령 사이에 세미 콜론을 넣으면 된다.



```

root@localhost:~# echo "My name is beomtaek" | gawk '{ $4="GilDong"; print $0 }'
My name is GilDong
root@localhost:~# gawk '{
> $4="beomtaek"
> print $0 }'
My name is GilDong
My name is beomtaek
  
```

## 기본 gawk (Cont.)

- 파일로부터 프로그램 읽기

gawk 편집기를 사용하면 파일에 프로그램을 저장하고 커맨드라인에서 이를 참조할 수 있다.

```

root@localhost ~]# cat script.gawk
{print $1 "'s home DIR is " $6 }
[root@localhost ~]#
[root@localhost ~]# gawk -F: -f script.gawk /etc/passwd
root's home DIR is /root
bin's home DIR is /bin
daemon's home DIR is /sbin
adm's home DIR is /var/adm
  
```

또는

```

root@localhost ~]# cat script2.gawk
{
text = "'s home DIR is "
print $1 text $6
}
[root@localhost ~]# gawk -F: -f script2.gawk /etc/passwd
root's home DIR is /root
bin's home DIR is /bin
daemon's home DIR is /sbin
adm's home DIR is /var/adm
lp's home DIR is /var/spool/lpd
  
```

## 기본 gawk (Cont.)

- 데이터 처리전에 스크립트 실행하기

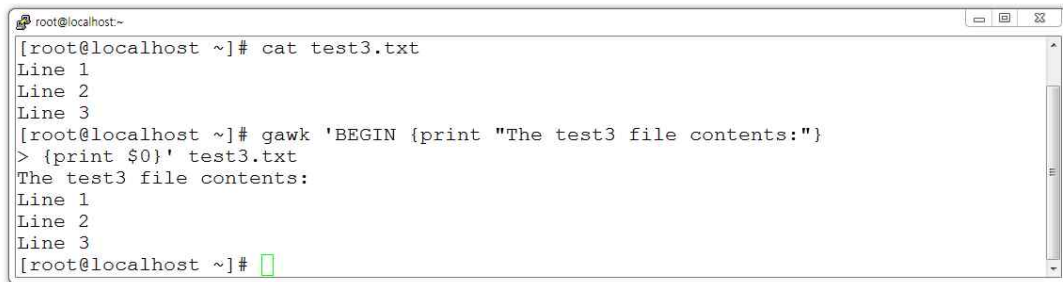
gawk 프로그램은 언제 프로그램의 스크립트를 실행할 수 있을지를 지정할 수 있다. 기본적으로 gawk는 입력 스트림을 통해 한 줄의 텍스트를 읽고 그 텍스트 줄의 데이터에 대해 프로그램 스크립트를 실행한다. 때로는 보고서의 머리말 부분을 만들 때 처럼 데이터를 처리하기 전에 스크립트를 실행할 수도 있다. BEGIN 키워드는 이러한 일을 위해 사용된다. 이 키워드는 gawk 가 데이터를 읽기 전에 지정된 프로그램 스크립트를 실행하도록 강제한다.

```

root@localhost ~]# gawk 'BEGIN {print "Hello world"}'
Hello world
[root@localhost ~]# gawk '{print "Hello world"}'
hi
Hello world
^C
[root@localhost ~]# █
  
```

## 기본 gawk (Cont.)

- 데이터 처리전에 스크립트 실행하기



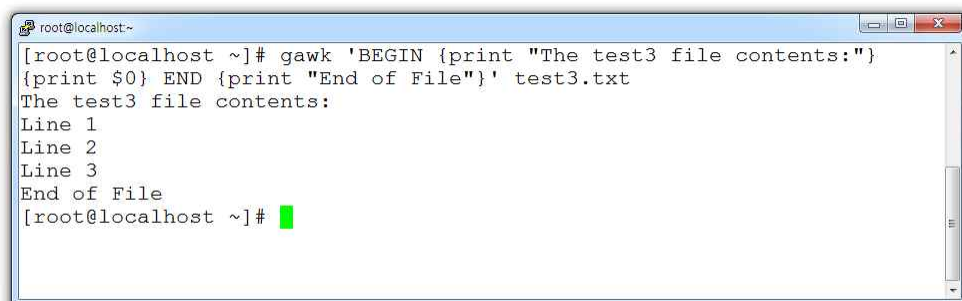
```

root@localhost:~
[root@localhost ~]# cat test3.txt
Line 1
Line 2
Line 3
[root@localhost ~]# gawk 'BEGIN {print "The test3 file contents:"}
> {print $0}' test3.txt
The test3 file contents:
Line 1
Line 2
Line 3
[root@localhost ~]#
  
```

## 기본 gawk (Cont.)

- 데이터를 처리한 후 스크립트 실행하기

BEGIN과 마찬가지로, END 키워드를 실행하면 gawk 가 데이터를 읽은 후에 실행할 프로그램을 지정할 수 있다.



```

root@localhost:~
[root@localhost ~]# gawk 'BEGIN {print "The test3 file contents:"}
{print $0} END {print "End of File"}' test3.txt
The test3 file contents:
Line 1
Line 2
Line 3
End of File
[root@localhost ~]#
  
```

## 정규표현식

- 정규 표현식의 정의

- 정규표현식은 리눅스 유틸리티가 텍스트를 걸러내는 데 사용하는 정의 패턴 템플릿
- 정규식 패턴은 데이터 스트림의 하나 이상의 문자를 뜻하는 와일드카드 문자 사용
- 정규표현식은 와일드 카드 패턴과 비슷한 방식으로 동작
- 정규표현식 엔진

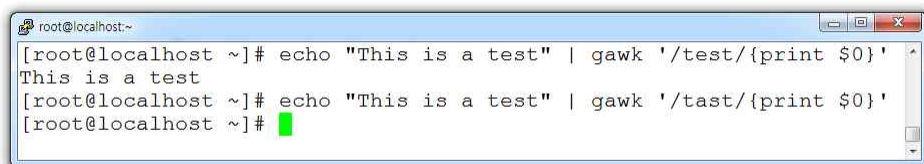
- 기본 정규 표현식
- 확장 정규 표현식

## 정규표현식 (Cont.)

- 기본 정규표현식

기본 정규표현식 패턴은 데이터 스트림에서 텍스트 문자와 대조된다.

- 일반 텍스트



```

root@localhost:~
[root@localhost ~]# echo "This is a test" | gawk '/test/{print $0}'
This is a test
[root@localhost ~]# echo "This is a test" | gawk '/tast/{print $0}'
[root@localhost ~]#
  
```

## 정규표현식 (Cont.)

- 특수문자

정규표현식 패턴은 몇 가지 글자에 특별한 의미를 부여한다. 텍스트 패턴에 이러한 문자를 사용하려고 하면 기대했던 결과를 얻을 수 없다. 이러한 특수 문자는 정규표현식으로 인식된다.

`.*[ ]^{}W+?#`

특수 문자 중 하나를 텍스트 문자처럼 사용하려고 하면 이스케이프시켜야 한다. 특수 문자를 이스케이프할 때에는 그 앞에 특수 문자를 두어서 이를 해석하는 정규표현식 엔진이 다음 글자가 일반 텍스트 문자라는 사실을 알게 해야 한다. 이러한 기능을 하는 특수 문자는 백슬래시 문자(`\`)이다.

## 정규표현식 (Cont.)

- gawk 에서의 확장 정규표현식

확장 정규표현식을 gawk 는 인식하지만 sed 는 인식하지 못한다.

- 물음표

물음표는 별표와 약간의 차이가 있다. 물음표는 그 앞에 있는 문자가 없거나 한번 나타나는 것을 뜻한다

```

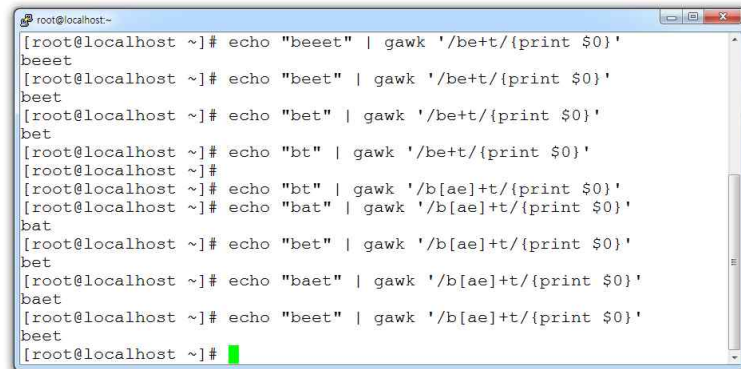
root@localhost ~# echo "bt" | gawk '/be?t/{print $0}'
bt
root@localhost ~# echo "bet" | gawk '/be?t/{print $0}'
bet
root@localhost ~# echo "beet" | gawk '/be?t/{print $0}'
beet
root@localhost ~# echo "beeet" | gawk '/be?t/{print $0}'
beeet
root@localhost ~# echo "bat" | gawk '/b[ae]?t/{print $0}'
bat
root@localhost ~# echo "bet" | gawk '/b[ae]?t/{print $0}'
bet
root@localhost ~# echo "baet" | gawk '/b[ae]?t/{print $0}'
baet
root@localhost ~#
  
```

## 정규표현식 (Cont.)

- gawk 에서의 확장 정규표현식

- + 기호

앞의 문자가 한 번 이상 나타날 수 있지만 한번 이상 있어야 한다. 즉, 문자가 아예 없다면 패턴은 일치하지 않는다.



```

[root@localhost ~]# echo "beeet" | gawk '/be+t/{print $0}'
beeet
[root@localhost ~]# echo "beet" | gawk '/be+t/{print $0}'
beet
[root@localhost ~]# echo "bet" | gawk '/be+t/{print $0}'
bet
[root@localhost ~]# echo "bt" | gawk '/be+t/{print $0}'
[root@localhost ~]#
[root@localhost ~]# echo "bt" | gawk '/b[ae]+t/{print $0}'
[root@localhost ~]# echo "bat" | gawk '/b[ae]+t/{print $0}'
bat
[root@localhost ~]# echo "bet" | gawk '/b[ae]+t/{print $0}'
bet
[root@localhost ~]# echo "baet" | gawk '/b[ae]+t/{print $0}'
baet
[root@localhost ~]# echo "beet" | gawk '/b[ae]+t/{print $0}'
beet
[root@localhost ~]#
  
```

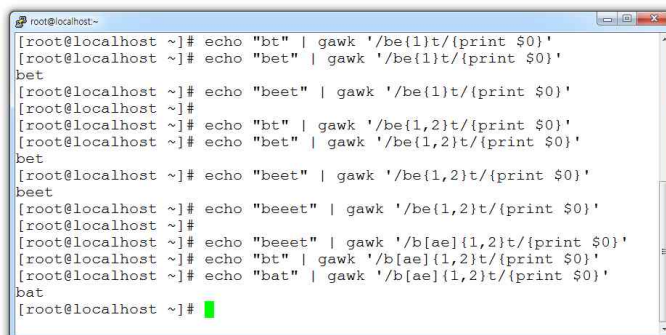
## 정규표현식 (Cont.)

- gawk 에서의 확장 정규표현식

- 중괄호 사용하기

인터벌이라고 하며, 다음의 두가지 형식이 있다

- m : 정규표현식이 정확히 m 번 일어남
- m,n : 정규표현식이 적어도 m, 최대 n 번 나타남



```

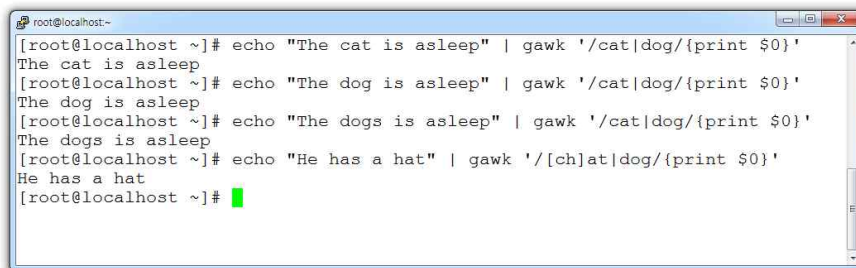
[root@localhost ~]# echo "bt" | gawk '/be{1}t/{print $0}'
[root@localhost ~]# echo "bet" | gawk '/be{1}t/{print $0}'
bet
[root@localhost ~]# echo "beet" | gawk '/be{1}t/{print $0}'
[root@localhost ~]#
[root@localhost ~]# echo "bt" | gawk '/be{1,2}t/{print $0}'
[root@localhost ~]# echo "bet" | gawk '/be{1,2}t/{print $0}'
bet
[root@localhost ~]# echo "beet" | gawk '/be{1,2}t/{print $0}'
beet
[root@localhost ~]# echo "beeet" | gawk '/be{1,2}t/{print $0}'
[root@localhost ~]#
[root@localhost ~]# echo "beeet" | gawk '/b[ae]{1,2}t/{print $0}'
[root@localhost ~]# echo "bt" | gawk '/b[ae]{1,2}t/{print $0}'
[root@localhost ~]# echo "bat" | gawk '/b[ae]{1,2}t/{print $0}'
bat
[root@localhost ~]#
  
```

## 정규표현식 (Cont.)

- gawk 에서의 확장 정규표현식

- 파이프 기호

파이프 기호는 데이터 스트림을 검사할 때 정규표현식 엔진이 논리 OR 식에 사용할 두 개 또는 그 이상의 패턴을 지정할 수 있다. 패턴 중 어느 것이든 데이터 스트림의 텍스트와 일치한다면 텍스트는 패턴 일치를 통과한다. 패턴 중 어느 것도 일치하지 않는다면 데이터 스트림 텍스트는 실패한다.



```

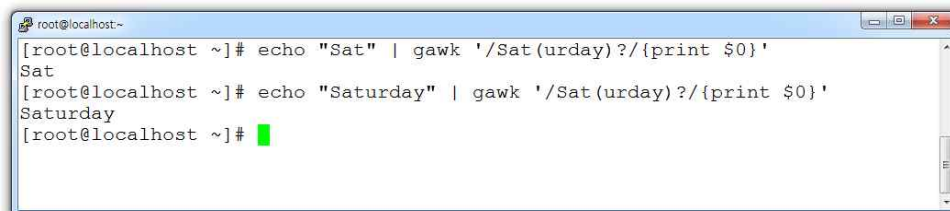
root@localhost:~# echo "The cat is asleep" | gawk '/cat|dog/{print $0}'
The cat is asleep
root@localhost:~# echo "The dog is asleep" | gawk '/cat|dog/{print $0}'
The dog is asleep
root@localhost:~# echo "The dogs is asleep" | gawk '/cat|dog/{print $0}'
The dogs is asleep
root@localhost:~# echo "He has a hat" | gawk '/[ch]at|dog/{print $0}'
He has a hat
root@localhost:~#
  
```

## 정규표현식 (Cont.)

- gawk 에서의 확장 정규표현식

- 표현식 그룹화 하기

괄호를 이용하여 그룹화 할 수 있다. 이를 통해 표준 문자처럼 취급할 수 있다.



```

root@localhost:~# echo "Sat" | gawk '/Sat(urday)?/{print $0}'
Sat
root@localhost:~# echo "Saturday" | gawk '/Sat(urday)?/{print $0}'
Saturday
root@localhost:~#
  
```



## 정규표현식 (Cont.)

- 예제문제
  - 사용자로 부터 입력받은 이메일 주소의 유효성을 검사하라

## 정규표현식 (Cont.)

- 예제문제 풀이

```
[root@localhost shell]# ./1.sh
이름을 입력해 주세요: 홍길동
이메일 주소를 입력해 주세요: gildong@gmail.com
당신의 이름은 홍길동 이고, 이메일 주소는 gildong@gmail.com 입니다.
해당 내용을 저장합니다
[root@localhost shell]# cat email.txt
김범택 bt78kim@gmail.com
홍길동 gildong@gmail.com
[root@localhost shell]#
[root@localhost shell]# ./1.sh
이름을 입력해 주세요: 김철수
이메일 주소를 입력해 주세요: chulsoo@gmail
./1.sh: line 11: [: chulsoo@gmail: unary operator expected
잘못된 이메일 형식입니다. 다시 입력하세요
[root@localhost shell]#
[root@localhost shell]# cat 1.sh
#!/bin/bash

echo -n "이름을 입력해 주세요: "
read name

echo -n "이메일 주소를 입력해 주세요: "
read email

check=$(echo $email | gawk '/^[a-z0-9_+.-]+@[a-z0-9-]+\.[a-z0-9]{2,4}$/ {print $0}')

if [ $email = $check ]
then
    echo "당신의 이름은 $name 이고, 이메일 주소는 $email 입니다."
    echo "해당 내용을 저장합니다"
    echo -e "$name\t$email" >> email.txt
else
    echo "잘못된 이메일 형식입니다. 다시 입력하세요"
fi
[root@localhost shell]#
```

## 구조적 명령과 gawk

- if 문

gawk 에서는 표준 if-then-else 형식의 if 문을 지원한다. 일반적인 형식은 아래와 같다

```
if (condition)
    statement1
```

또한 다음과 같이 한 줄에 작성할 수도 있다.

```
if (condition) statement1
```

```
[root@localhost gawk]# cat data4
10
5
13
50
34
[root@localhost gawk]#
[root@localhost gawk]# gawk '{if ($1 > 20) print $1}' data4
50
34
[root@localhost gawk]#
```

## 구조적 명령과 gawk (Cont.)

- if 문

if문에서 여러 명령을 실행해야 한다면 중괄호로 묶어야 한다.

```
[root@localhost gawk]# gawk '{
> if ( $1 > 20 )
> {
>   x = $1 * 2
>   print x
> }
> }' data4
100
68
[root@localhost gawk]# gawk '{
> if ( $1 > 20 )
> {
>   x = $1 * 2
>   print x
> } else
> {
>   x = $1 / 2
>   print x
> } }' data4
5
2.5
6.5
100
68
[root@localhost gawk]#
```

## 구조적 명령과 gawk (Cont.)

- if 문

else 구문을 같은 줄에 쓸 수도 있지만 if 구문 부분 다음에 세미콜론을 사용해야 한다.

```
if (condition) statement1; else statement2
```

다음은 한 줄 형식을 사용한 예이다.

```
[root@localhost gawk]# gawk '{if ($1 > 20) print $1 * 2; else print $1 / 2}' data4
5
2.5
6.5
100
68
[root@localhost gawk]#
```

## 구조적 명령과 gawk (Cont.)

- for 문

gawk 프로그래밍 언어는 C 언어 스타일의 루프를 지원한다.

```
for( variable 지정; condition; iteration process)
```

```
[root@localhost gawk]# gawk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
>   total += $i
> }
> avg = total / 3
> print "Average:", avg
> }' data5
Average: 128.333
Average: 137.667
Average: 176.667
[root@localhost gawk]#
```

## 구조적 명령과 gawk (Cont.)

- while 문

while 문은 gawk 프로그램에 기본 루프 기능을 제공한다. 다음은 기본 형식이다.

```
while (condition)
{
    statements
}
```

while 루프는 데이터의 세트를 차례대로 되풀이하면서 반복 작업을 중단할 조건을 검사한다. 각 레코드마다 계산에 사용할 여러 가지 데이터 값이 있을 때 유용하다

```
[root@localhost gawk]# cat data5
130    120    135
160    113    140
145    170    215

[root@localhost gawk]# gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
>     total += $i
>     i++
> }
> avg = total / 3
> print "Average:", avg
> }' data5
Average: 128.333
Average: 137.667
Average: 176.667
[root@localhost gawk]#
```

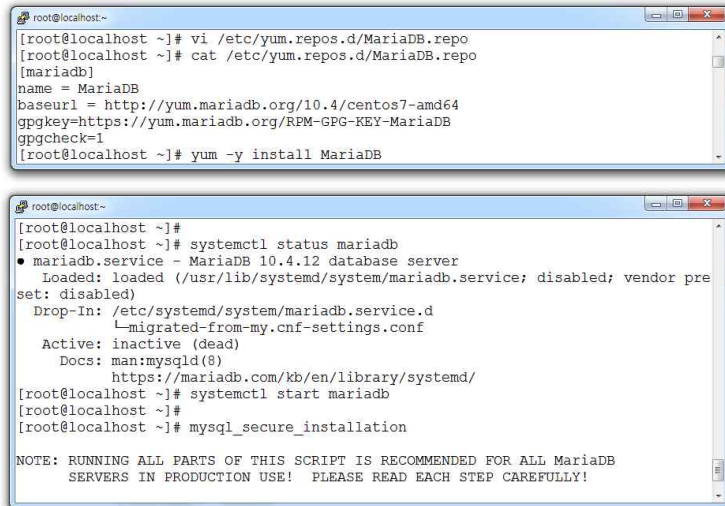
## 데이터베이스와 함께 사용하기

- 교육내용

- 기본 MariaDB 사용하기
- 스크립트에서 데이터베이스 사용하기

## 기본 mariaDB 사용하기

- mariaDB 설치 및 연결하기



```

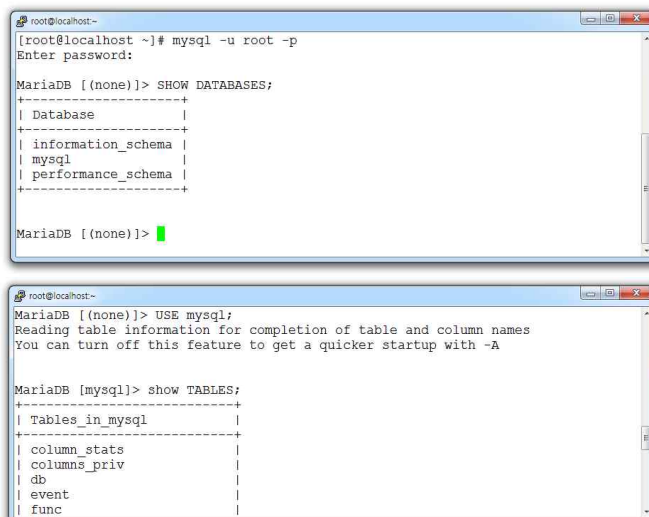
root@localhost~# vi /etc/yum.repos.d/MariaDB.repo
root@localhost ~# cat /etc/yum.repos.d/MariaDB.repo
[mariadb]
name = MariaDB
baseurl = http://yum.mariadb.org/10.4/centos7-amd64
gpgkey=https://yum.mariadb.org/RPM-GPG-KEY-MariaDB
gpgcheck=1
root@localhost ~# yum -y install MariaDB

root@localhost~#
root@localhost ~# systemctl status mariadb
● mariadb.service - MariaDB 10.4.12 database server
   Loaded: loaded (/usr/lib/systemd/system/mariadb.service; disabled; vendor pre
  set: disabled)
   Drop-In: /etc/systemd/system/mariadb.service.d
            └─migrated-from-my.cnf-settings.conf
   Active: inactive (dead)
     Docs: man:mysqld(8)
           https://mariadb.com/kb/en/library/systemd/
root@localhost ~# systemctl start mariadb
root@localhost ~#
root@localhost ~# mysql_secure_installation

NOTE: RUNNING ALL PARTS OF THIS SCRIPT IS RECOMMENDED FOR ALL MariaDB
SERVERS IN PRODUCTION USE! PLEASE READ EACH STEP CAREFULLY!
  
```

## 기본 mariaDB 사용하기 (Cont.)

- mariaDB 설치 및 연결하기



```

root@localhost~# mysql -u root -p
Enter password:

MariaDB [(none)]> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
+-----+

MariaDB [(none)]>

root@localhost~#
MariaDB [(none)]> USE mysql;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

MariaDB [mysql]> show TABLES;
+-----+
| Tables_in_mysql |
+-----+
| column_stats |
| columns_priv |
| db |
| event |
| func |
+-----+
  
```

## 기본 mariaDB 사용하기 (Cont.)

- Database, 사용자, 테이블 생성

데이터베이스 생성하기

```
MariaDB [(none)]> CREATE DATABASE mytest;
Query OK, 1 row affected (0.002 sec)

MariaDB [(none)]> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| mytest |
| performance_schema |
+-----+
4 rows in set (0.002 sec)

MariaDB [(none)]> █
```

## 기본 mariaDB 사용하기 (Cont.)

- Database, 사용자, 테이블 생성

사용자 계정 생성하기 (아래 참고하여 직접 생성해 볼 것)

```
MariaDB [(none)]> GRANT SELECT,INSERT,DELETE,UPDATE ON mytest.* TO test IDENTIFIED b
y 'test';
Query OK, 0 rows affected (0.002 sec)

[root@localhost ~]# mysql mytest -u test -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 21
Server version: 10.4.12-MariaDB MariaDB Server

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [mytest]> CREATE TABLE employees (
-> empid int not null,
-> lastname varchar(30),
-> firstname varchar(30),
-> salary float,
-> primary key (empid));
ERROR 1142 (42000): CREATE command denied to user 'test'@'localhost' for table 'employees'
MariaDB [mytest]> █
```

## 기본 mariaDB 사용하기 (Cont.)

- Database, 사용자, 테이블 생성

### 테이블 생성하기

```
MariaDB [(none)]> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mytest    |
+-----+
2 rows in set (0.001 sec)

MariaDB [(none)]> use mytest;
Database changed
MariaDB [mytest]> CREATE TABLE employees ( empid int not null, lastname varchar(30), first
name varchar(30), salary float, primary key (empid));
Query OK, 0 rows affected (0.007 sec)

MariaDB [mytest]> █
```

## 기본 mariaDB 사용하기 (Cont.)

- Database, 사용자, 테이블 생성

### 데이터 삽입 및 삭제

테이블에 새로운 데이터를 삽입하려면 INSERT SQL 명령을 사용한다. 각 INSERT 명령은 DB서버가 레코드를 받아들이기 위한 데이터필드 값을 지정해야 한다

일반적인 형식은 다음과 같다 "INSERT INTO table VALUES (...)"

```
MariaDB [mytest]> INSERT INTO employees VALUES (1, 'HONG', 'GILDONG', 3000);
Query OK, 1 row affected (0.007 sec)

MariaDB [mytest]> INSERT INTO employees VALUES (2, 'KIM', 'CHULS00', 2800 );
Query OK, 1 row affected (0.002 sec)

MariaDB [mytest]> INSERT INTO employees VALUES (3, 'PARK', 'YOUNGHEE', 3200 );
Query OK, 1 row affected (0.036 sec)

MariaDB [mytest]> INSERT INTO employees VALUES (4, 'LEE', 'JISEON', 2400 );
Query OK, 1 row affected (0.003 sec)

MariaDB [mytest]> █
```

## 기본 mariaDB 사용하기 (Cont.)

- Database, 사용자, 테이블 생성

데이터 삽입 및 삭제

테이블에서 데이터를 제거할 때에는 DELETE SQL 명령을 사용한다. 기본 DELETE 명령 형식

```
DELETE FROM table;
```

하나의 레코드 또는 그룹을 삭제하고자 할 경우에는 WHERE 절을 사용해야 한다.

```
MariaDB [mytest]> INSERT INTO employees VALUES (4, 'LEE', 'JISEON', 2400 );
Query OK, 1 row affected (0.003 sec)
```

```
MariaDB [mytest]> DELETE FROM employees WHERE empid = 4;
Query OK, 1 row affected (0.005 sec)
```

```
MariaDB [mytest]> █
```

## 기본 mariaDB 사용하기 (Cont.)

- Database, 사용자, 테이블 생성

데이터 질의

작성된 데이터를 확인하고 싶다면 SELECT 를 사용할 수 있다. 일반적인 SELECT 문의 기본형식

```
SELECT 데이터필드 FROM table
```

```
MariaDB [mytest]> SELECT * FROM employees;
+-----+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+-----+
| 1 | HONG | GILDONG | 3000 |
| 2 | KIM | CHULSOO | 2800 |
| 3 | PARK | YOUNGHEE | 3200 |
+-----+-----+-----+-----+
3 rows in set (0.002 sec)
```

```
MariaDB [mytest]> █
```



## 기본 mariaDB 사용하기 (Cont.)

- Database, 사용자, 테이블 생성

데이터 질의

데이터베이스 서버는 질의된 요청에 대하여 데이터를 돌려주는 방법을 정의하는 몇가지 변경자를 제공한다.

- WHERE : 특정 조건을 충족하는 레코드들의 부분집합을 표시
- ORDER BY : 지정된 순서로 레코드를 표시
- LIMIT : 레코드들의 부분집합을 표시

```
MariaDB [mytest]> SELECT * FROM employees WHERE salary > 3000;
+-----+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+-----+
| 3 | PARK | YOUNGHEE | 3200 |
+-----+-----+-----+-----+
1 row in set (0.003 sec)
```

```
MariaDB [mytest]> exit
Bye
[root@localhost ~]# cd
[root@localhost ~]# ls
```

## 스크립트에서 데이터베이스 사용하기

- 서버에 명령 보내기

서버 연결을 설정하고 나면 데이터베이스와 상호작용하는 명령을 보낼 수 있다. 이러한 작업을 하는 방법은 두 가지가 있다.

- 하나의 명령을 보내고 종료
- 여러 개의 명령을 보냄

하나의 명령을 보내려면 mysql 커맨드라인의 일부로 명령을 포함시켜야 한다. mysql 명령에 대해서는 -e (execute) 매개변수를 사용하여 이러한 작업을 할 수 있다.

```
[root@localhost ~]# cat mtest1
#!/bin/bash

MYSQL=$(which mysql)

$MYSQL mytest -u test -ptest -e 'select * from employees'
[root@localhost ~]#
[root@localhost ~]# ./mtest1
+-----+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+-----+
| 1 | HONG | GILDONG | 3000 |
| 2 | KIM | CHULSOO | 2800 |
| 3 | PARK | YOUNGHEE | 3200 |
+-----+-----+-----+-----+
[root@localhost ~]#
```

## 스크립트에서 데이터베이스 사용하기 (Cont.)

- 서버에 명령 보내기

다음은 파일 끝 문자열을 지정하는 방법으로 그 사이에 데이터가 들어 있다.

```
[root@localhost ~]# cat mtest2
#!/bin/bash

MYSQL=$(which mysql)

$MYSQL mytest -u test -ptest <<EOF
show tables;
select * from employees where salary > 3000;
EOF
[root@localhost ~]# ./mtest2
Tables_in_mytest
employees
empid  lastname      firstname      salary
3      PARK          YOUNGHEE      3200
[root@localhost ~]#
```

## 스크립트에서 데이터베이스 사용하기 (Cont.)

- 서버에 명령 보내기

테이블에서 데이터를 가져오는 것에만 국한되는 것은 아니다. 스크립트에서 INSERT 문을 비롯해서 어떤 종류의 SQL 명령이든 사용할 수 있다.

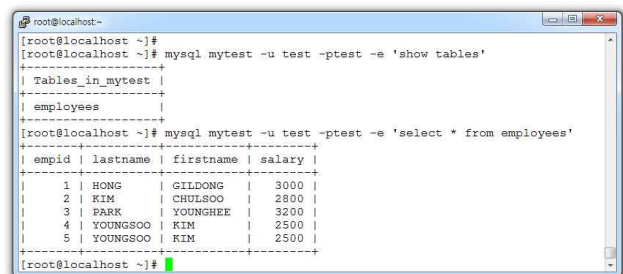
```
[root@localhost ~]# cat mtest3
#!/bin/bash

MYSQL=$(which mysql)
if [ $# -ne 4 ]
then
echo "Usage : mtest3 직원번호 이름 성 연봉"
else
statement="INSERT INTO employees VALUES ($1, '$2', '$3', $4)"
$MYSQL mytest -u test -ptest << EOF
$statement
EOF

if [ $? -eq 0 ]
then
echo "데이터가 성공적으로 작성되었습니다"
else
echo "데이터 추가에 문제가 발생하였습니다"
fi

fi

[root@localhost ~]# ./mtest3
Usage : mtest3 직원번호 이름 성 연봉
[root@localhost ~]#
[root@localhost ~]# ./mtest3 4 YOUNGSOO KIM 2500
ERROR 1062 (23000) at line 1: Duplicate entry '4' for key 'PRIMARY'
데이터 추가에 문제가 발생하였습니다
[root@localhost ~]#
[root@localhost ~]# ./mtest3 5 YOUNGSOO KIM 2500
데이터가 성공적으로 작성되었습니다
[root@localhost ~]#
```



```
[root@localhost ~]#
[root@localhost ~]# mysql mytest -u test -ptest -e 'show tables'
+-----+
| Tables_in_mytest |
+-----+
| employees        |
+-----+
[root@localhost ~]# mysql mytest -u test -ptest -e 'select * from employees'
+-----+
| empid | lastname | firstname | salary |
+-----+
| 1      | HONG     | GILDONG   | 3000   |
| 2      | KIM      | CHULSOO   | 2800   |
| 3      | PARK     | YOUNGHEE  | 3200   |
| 4      | YOUNGSOO | KIM       | 2500   |
| 5      | YOUNGSOO | KIM       | 2500   |
+-----+
```

## 리눅스 셸 프로그래밍

---

### 고급부분

#### 함수만들기

- 교육내용
  - 기본 스크립트 함수
  - 값을 돌려주기
  - 함수에서 변수 사용하기
  - 배열 및 가변 함수
  - 재귀함수
  - 라이브러리 만들기

## 기본 스크립트 함수

- 함수는 이름을 지정하고 코드에서 다시 사용할 수 있는 스크립트 코드의 블록이다. 함수로 묶어 놓은 스크립트 코드의 블록을 사용해야 한다면 지정해 놓은 함수의 이름을 사용하면 된다(함수호출).

- 함수 생성 두가지 형식

```
function name {
  commands
}
```

- (위) 코드 블록에 함수 이름을 지정하는 키워드 함수. Name 속성은 함수에 지정할 고유한 이름을 정의한다. 스크립트에서 사용자가 정의하는 함수에는 각각 고유한 이름을 지정해야 한다. Commands 는 함수를 구성하는 하나 이상의 bash 셸 명령이다. 함수를 호출할 때 bash 셸은 보통의 스크립트 처럼 함수에 나타나는 순서로 각 명령을 실행한다.
- (아래) 다른 프로그래밍 언어에서 함수를 정의하는 방법에 가깝다. 함수 이름 뒤에 있는 빈 괄호는 함수를 정의하는 것이다. 원래의 쉘 스크립트 함수 형식과 같은 이름 지정 규칙이 적용된다.

```
name() {
  commands
}
```

## 기본 스크립트 함수 (Cont.)

- 함수 사용하기

```
[root@srv1 test]# cat test1.sh
#!/bin/bash

function func1 {
    echo "first example"
}

count=1
while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done

echo "This is the end of the loop"
func1
echo "Now this is the end of the script"
[root@srv1 test]# ./test1.sh
first example
first example
first example
first example
first example
This is the end of the loop
first example
Now this is the end of the script
[root@srv1 test]#
```

## 기본 스크립트 함수 (Cont.)

- 함수 사용하기

```
[root@srv1 test]# cat test3.sh
#!/bin/bash

function func1 {
    echo "This is the first definition of the function name"
}

func1

function func1 {
    echo "This is a repeat of the same function name"
}

func1
echo "This is the end of the script"
[root@srv1 test]#
[root@srv1 test]# ./test3.sh
This is the first definition of the function name
This is a repeat of the same function name
This is the end of the script
[root@srv1 test]# █
```

## 값을 돌려주기

- 기본 종료상태

- 기본적으로 함수의 종료 상태는 함수의 마지막 명령이 돌려주는 종료 상태다. 함수가 실행된 후에 종료 상태를 확인하기 위한 표준 \$? 변수를 사용할 수 있을까?

```
[root@srv1 test]# cat test4.sh
#!/bin/bash

func1() {
    echo "trying to display a non-existent file"
    ls -l badfile
}

echo "testing the function: "
func1

echo "The exit status is: $?"
[root@srv1 test]#
[root@srv1 test]# ./test4.sh
testing the function:
trying to display a non-existent file
ls: cannot access badfile: No such file or directory
The exit status is: 2
[root@srv1 test]#
```

```
[root@srv1 test]# cat test4b.sh
#!/bin/bash

func1() {
    ls -l badfile
    echo "This was a test of a bad command"
}

echo "testing the function:"
func1
echo "The exit status is: $?"

[root@srv1 test]#
[root@srv1 test]# ./test4b.sh
testing the function:
ls: cannot access badfile: No such file or directory
This was a test of a bad command
The exit status is: 0
[root@srv1 test]# █
```

## 값을 돌려주기 (Cont.)

- return 명령 사용하기

- bash 셸은 함수가 특정한 종료 상태를 돌려줄 수 있도록 return 명령을 사용한다. return 함수로 종료 상태를 정의하는 단일한 정수 값을 지정할 수 있으며 함수의 종료 상태를 프로그래밍으로 설정하는 손쉬운 방법을 제공한다.

```
[root@srv1 test]# cat test5
#!/bin/bash

function dbl {
    read -p "Enter a value: " value
    echo "doubling the value"
    return ${value * 2}
}

dbl
echo "The new value is $?"
[root@srv1 test]#
[root@srv1 test]# ./test5
Enter a value: 10
doubling the value
The new value is 20
[root@srv1 test]#
```

- 함수가 완료되었을 때 될 수 있는 대로 빨리 값을 저장하기
- 종료 상태는 0에서 255의 범위 안에 있어야 함
- 큰 정수 값 또는 문자열 값을 돌려줘야 한다면 이러한 방법을 사용할 수 없다. 대신 다음 섹션에서 설명할 다른 방법을 사용해야 한다.

## 값을 돌려주기 (Cont.)

- 함수출력 사용하기

- 명령의 출력을 쉘 변수에 저장할 수 있는 것처럼 함수의 출력도 쉘 변수에 저장할 수 있다. 변수에서 나온 어떤 유형의 출력이라도 변수에 할당하면 된다.

```
[root@srv1 test]# cat test5b
#!/bin/bash

function dbl {
    read -p "Enter a value: " value
    echo ${value * 2}
}

result=$(dbl)
echo "The new value is $result"

[root@srv1 test]# ./test5b
Enter a value: 10
The new value is 20
[root@srv1 test]#
```

- 새로운 함수는 이제 계산 결과를 표시하는 echo 문을 사용한다. 스크립트는 답을 받기 위해 종료상태 대신 dbl 함수의 출력을 저장

## 함수에서 변수 사용하기

- 함수에 매개변수 전달하기

- 앞의 '값 돌려주기' 절에서 언급한 바와 같이 bash 셸은 함수를 미니 스크립트 처럼 다룬다. 이는 보통의 스크립트처럼 함수에도 매개변수를 전달할 수 있다는 것을 뜻한다.

함수는 커맨드라인에서 함수에 전달되는 매개변수를 대신하기 위한 표준 매개변수 환경 변수를 사용할 수 있다. 예를 들어, 함수의 이름은 \$0 변수로 정의되고 함수의 커맨드라인에 있는 모든 매개변수는 \$1, \$2 와 같은 변수로 정의된다. 또한 함수에 전달된 매개변수의 수를 결정하기 위한 특수한 변수 \$# 도 사용할 수 있다.

스크립트에서 함수를 지정할 때에는 함수와 같은 커맨드라인에 매개변수를 제공해야 한다. 예) func1 \$value1 10

## 함수에서 변수 사용하기 (Cont.)

- 함수에 매개변수 전달하기

- 다음은 함수에 값을 전달하기 위해 앞서 언급한 방법을 사용한 예 이다

```
[root@srv1 test]# cat test6
#!/bin/bash

function addem {
    if [ $# -eq 0 ] || [ $# -gt 2 ]
    then
        echo -1
    elif [ $# -eq 1 ]
    then
        echo $[ $1 + $1 ]
    else
        echo $[ $1 + $2 ]
    fi
}

echo -n "Adding 10 and 15 : "
value=$(addem 10 15)
echo $value
echo -n "Let's try adding just one number: "
value=$(addem 10)
echo $value
echo -n "Now trying adding no numbers: "
value=$(addem)
echo $value
echo -n "finally, try adding three numbers: "
value=$(addem 10 15 20)
echo $value
[root@srv1 test]# ./test6
Adding 10 and 15 : 25
Let's try adding just one number: 20
Now trying adding no numbers: -1
finally, try adding three numbers: -1
[root@srv1 test]#
```

addem 함수는 먼저 스크립트가 전달한 매개변수의 수를 확인한다.  
매개변수가 없거나 매개변수가 두 개보다 많다면 addem 함수는 -1 을 돌려준다.  
매개변수가 하나만 있다면, addem 함수는 이 매개변수를 두 배로 만들어서  
결과값으로 사용한다. 매개변수가 두 개 있다면 addem 함수는 두 변수를 더해서  
결과값으로 사용한다.

## 함수에서 변수 사용하기 (Cont.)

- 함수에 매개변수 전달하기

- 함수가 변수 \$1, \$2를 사용한다고 해도 이들 변수는 스크립트의 메인에서 쓰는 \$1, \$2 변수와는 다르다. 함수에서 이들 값을 사용하려면 함수를 호출할 때 직접 값을 넘겨주어야 한다.

```
[root@srv1 test]# cat test7
#!/bin/bash

function func7 {
    echo $[ $1 * $2 ]
}

if [ $# -eq 2 ]
then
    value=$(func7 $1 $2)
    echo "The result is $value"
else
    echo "Usage: badtest1 a b"
fi

[root@srv1 test]# ./test7
Usage: badtest1 a b
[root@srv1 test]#
[root@srv1 test]# ./test7 10 20
The result is 200
[root@srv1 test]#
```

- 함수에 \$1, \$2 변수를 넘겨줌으로써, 다른 매개변수처럼 이들 값을 함수에서 쓸 수 있게 된다.

## 함수에서 변수 사용하기 (Cont.)

- 함수에서 변수 다루기

- 함수는 두 종류의 변수를 사용한다. 전역(global)변수, 지역(local)변수

- 전역변수

전역변수는 쉘 스크립트 안이라면 어디서나 쓸 수 있는 변수다. 스크립트의 메인 부분에서 전역 변수를 정의했다면 함수 안에서 이 값을 검색할 수 있다. 마찬가지로, 함수 안에서 전역 변수를 정의하면 스크립트의 메인 섹션에서 이 값을 검색할 수 있다.

기본적으로 스크립트에서 정의한 모든 변수는 전역변수다. 함수의 외부에서 정의된 변수는 함수 안에서 사용할 수 있다.

```
[root@srv1 test]# cat test8
#!/bin/bash

function dbl {
    value=$[ $value * 2 ]
}

read -p "Enter a value: " value
dbl
echo "The new value is: $value"

[root@srv1 test]#
[root@srv1 test]# ./test8
Enter a value: 250
The new value is: 500
[root@srv1 test]#
```

- \$value 변수는 함수 밖에서 정의되었고 함수 밖에서 값이 할당되었다. dbl 함수가 호출될 때 변수 및 그 값은 함수 안에서도 계속 유효하다. 변수가 함수 안에서 새 값을 할당하면 스크립트가 변수를 참조할 때 새로 할당된 값이 계속 유효하다. 하지만 이는 위험할 수 있다. 특히 함수를 다른 쉘 스크립트에서 사용하려면 더더욱 위험하다. 전역 변수는 어떤 변수가 함수 안에서 쓰이고 있는지 정확히 알아야 한다. 여기에는 함수 내부에서 계산을 위해서만 쓰이고 스크립트로 값을 돌려주지 않는 모든 변수도 포함된다.



## 함수에서 변수 사용하기 (Cont.)

- 함수에서 변수 다루기

- 앞서 언급한 문제점이 발생할 수 있는 예

```
[root@srv1 test]# cat badtest8
#!/bin/bash

function func1 {
    temp=$(( $value + 5 ))
    result=$(( $temp * 2 ))
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
[root@srv1 test]# ./badtest8
The result is 22
temp is larger
[root@srv1 test]#
```

- 함수에 \$temp 변수를 사용해서 스크립트 안에서 변수의 값이 변형되어 원치 않은 결과가 나왔다. 함수 안에서 일어날 수 있는 이러한 문제를 해결할 수 있는 손쉬운 방법은 다음 절에 있다

## 함수에서 변수 사용하기 (Cont.)

- 함수에서 변수 다루기

- 지역변수

함수에서 전역 변수를 사용하는 대신 함수 안에서 사용되는 모든 변수를 지역 변수로 선언할 수 있다. 이를 위해서는 변수 선언 앞에 local 키워드를 사용하면 된다.

변수에 값을 할당하는 동안 할당문에서 local 키워드를 사용할 수 있다 "local temp=\$(( \$value + 5 ))"

```
[root@srv1 test]# cat test9
#!/bin/bash

function func1 {
    local temp=$(( $value + 5 ))
    result=$(( $temp * 2 ))
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
[root@srv1 test]# ./test9
The result is 22
temp is smaller
[root@srv1 test]#
```

func1 함수 안에서 \$temp 변수를 사용해도 메인 스크립트에 있는 \$temp 변수에 할당된 값에는 영향을 주지 않는다

## 배열 변수와 함수

### • 함수에 배열 전달하기

- 환경 변수의 유용한 기능은 이들을 배열로 사용할 수 있다는 것이다. 배열은 여러 값을 저장할 수 있는 변수다. 값은 개별적으로 또는 전체 배열을 통째로 참조할 수 있다.

환경 변수에 여러 값을 사용하려면 괄호 안에 값을 넣고 각각의 값은 빈 칸으로 구분하면 된다

```
[root@localhost ~]# mytest=(one two three four five)
[root@localhost ~]#
[root@localhost ~]# echo $mytest
one
[root@localhost ~]# echo ${mytest[0]}
one
[root@localhost ~]# echo ${mytest[2]}
three
[root@localhost ~]# echo ${mytest[*]}
one two three four five
[root@localhost ~]#
[root@localhost ~]# #### 개인 인덱스 위치의 값을 변경할 수 있다 ####
[root@localhost ~]#
[root@localhost ~]# mytest[2]=seven
[root@localhost ~]# echo ${mytest[*]}
one two seven four five
[root@localhost ~]#
[root@localhost ~]# #### 배열 안의 개별값을 지우기 위해
[root@localhost ~]# #### unset 사용가능하지만 까다롭다
[root@localhost ~]#
[root@localhost ~]# unset mytest[2]
[root@localhost ~]# echo ${mytest[*]}
one two four five
[root@localhost ~]# echo ${mytest[2]}

[root@localhost ~]# echo ${mytest[3]}
four
[root@localhost ~]#
[root@localhost ~]# unset mytest
[root@localhost ~]# echo ${mytest[*]}
```

## 배열 변수와 함수 (Cont.)

### • 함수에 배열 전달하기

- 함수에 배열 변수 값을 사용하기는 조금 까다로우며, 몇 가지 고려해야 할 점들이 있다. 이 절에서는 함수에서 배열 변수를 사용하는 방법을 알아본다
- 스크립트 함수에 배열 변수를 전달하는 기법은 헛갈릴 수 있다. 단일 매개변수로 배열 변수를 전달하려고 하면 제대로 되지 않는다.

```
[root@srv1 test]# cat badtest3
#!/bin/bash

function testit {
    echo "The parameters are: $@"
    thisarray=$1
    echo "The received array is ${thisarray[*]}"
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
testit $myarray
[root@srv1 test]# ./badtest3
The original array is: 1 2 3 4 5
The parameters are: 1
The received array is 1
[root@srv1 test]#
```

- 함수 매개변수로 배열 변수를 쓰려면 함수는 배열 변수의 첫 번째 값만 가져온다. 이 문제를 해결하기 위해서는 배열 변수를 개별 값으로 분해해서 변수를 함수 매개변수로 사용해야 한다. 함수 안에서 새로운 배열 변수에 모든 매개변수를 재구성할 수 있다. 다음 페이지에 그 예가 나온다

## 배열 변수와 함수 (Cont.)

- 함수에 배열 전달하기

```
[root@srv1 test]# cat test10
#!/bin/bash

function testit {
    local newarray
    newarray=(`echo "$@"`)
    echo "The new array value is: ${newarray[*]}"
}

myarray=(1 2 3 4 5)
echo "The original array is ${myarray[*]}"
testit ${myarray[*]}
[root@srv1 test]#
[root@srv1 test]# ./test10
The original array is 1 2 3 4 5
The new array value is: 1 2 3 4 5
[root@srv1 test]#
```

## 배열 변수와 함수 (Cont.)

- 함수에 배열 전달하기

- 아래의 스크립트는 함수의 커맨드라인에 놓을 배열 변수의 모든 개별값을 가지고 있는 \$myarray 변수를 사용한다. 함수는 커맨드라인 매개변수로 부터 배열 변수를 다시 만든다. 함수 안에서 이 배열은 다른 배열처럼 사용할 수 있다.

```
[root@srv1 test]# cat test11
#!/bin/bash

function addarray {
    local sum=0
    local newarray
    newarray=(`echo "$@"`)
    for value in ${newarray[*]}
    do
        sum=$(( sum + $value ))
    done
    echo $sum
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1=(`echo ${myarray[*]}`)
result=$(addarray $arg1)
echo "The result is $result"
[root@srv1 test]#
[root@srv1 test]# ./test11
The original array is: 1 2 3 4 5
The result is 15
[root@srv1 test]#
```

- addarray 함수는 배열 값을 차례대로 되풀이하여, 이들 값을 모두 더한다. myarray 배열 변수에 어떤 값이든 넣으면 addarray 함수가 이들을 모두 더한다

## 배열 변수와 함수 (Cont.)

- 함수에서 배열 돌려주기

- 셸 스크립트로 함수에서 다시 배열 변수를 돌려줄 때 비슷한 방법을 사용한다. 함수는 개별 배열 값을 출력하기 위한 적절한 echo 문을 사용하고 스크립트는 이를 새로운 배열 변수로 재구성해야 한다.

```
[root@srv1 test]# cat test12
#!/bin/bash

function arraydbl {
    local origarray
    local newarray
    local elements
    local i
    origarray=( $(echo "$@") )
    newarray=( $(echo "$@" ) )
    elements=$(( $# - 1 ))
    for (( i = 0; i <= $elements; i++ ))
    {
        newarray[i]=$[ ${origarray[i]} * 2 ]
    }
    echo ${newarray[*]}
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1=$(echo ${myarray[*]})
result=$(arraydbl $arg1)
echo "The new array is: ${result[*]}"
[root@srv1 test]#
[root@srv1 test]# ./test12
The original array is: 1 2 3 4 5
The new array is: 2 4 6 8 10
[root@srv1 test]#
```

스크립트는 \$arg1 변수를 사용하여 배열 값을 arraydbl 함수에 전달된다. Arraydbl 함수는 전달 받은 배열 값으로 새로운 배열 변수를 재구성하고, 출력 배열 변수를 위한 복사본을 만든다. 그 다음 함수는 배열 값을 차례대로 되돌리하면서 각각의 값을 두 배로 만들고 나서는 함수 안의 배열 변수 복사본에 이를 저장한다.

arraydbl 함수는 배열 변수의 개별 값을 출력하기 위해 echo 문을 사용한다. 스크립트는 arraydbl 함수의 출력을 사용하여 새로운 배열 변수를 재구성한다.

## 재귀 함수

- 지역 함수 변수가 제공하는 기능 중 하나는 폐쇄성이다. 폐쇄성 함수는 스크립트가 커맨드라인을 통해 전달하는 값을 제외하고는 함수 바깥의 어떤 자원도 사용하지 않는다. 이러한 특징 때문에 함수는 재귀 호출될 수 있으며 이는 함수가 답을 얻을 때까지 스스로를 호출할 수 있다는 뜻이다.
- 보통 재귀함수는 마지막으로 어디까지 되풀이할 것인가를 정하는 베이스 값이 있다. 베이스 값이 정의한 조건에 다다를 때까지 복잡한 수식을 한 단계로 줄여서 되풀이하기 위해 재귀적 방법을 사용하는 고급 수학 알고리즘들이 많다.
- 재귀 알고리즘의 고전적인 예는 계승 계산이다. 계승이란 지정된 수의 앞에 있는 모든 수와 자기 자신을 곱한 것이다. 따라서 5의 계승을 찾기 위해서는 다음과 같은 계산을 한다.

$$5! = 1 * 2 * 3 * 4 * 5 = 120$$

재귀적 방법으로 위의 식은 다음과 같이 줄어든다

$$x! = x * (x - 1)!$$

## 재귀 함수 (Cont.)

- $x$ 의 계승은  $x-1$ 의 계승에  $x$ 를 곱한 것이다. 이는 간단한 재귀적 스크립트로 다음과 같이 표현한다
- factorial 함수는 값을 계산하기 위해서 factorial 자신을 사용한다

```
[root@srv1 test]# cat test13
#!/bin/bash

function factorial {
    if [ $1 -eq 1 ]
    then
        echo 1
    else
        local temp=$(( $1 - 1 ))
        local result=$(factorial $temp)
        echo $(( result * $1 ))
    fi
}

read -p "Enter value: " value
result=$(factorial $value)
echo "The factorial of $value is: $result"
[root@srv1 test]# ./test13
Enter value: 5
The factorial of 5 is: 120
[root@srv1 test]#
```

## 라이브러리 만들기

- 함수가 하나의 스크립트 안에서 입력에 들어가는 노력을 절약한다는 것은 쉽게 알 수 있다. 하지만 여러 스크립트가 같은 코드 블록을 사용하는 일이 있다면? 각 스크립트 마다 같은 함수를 매번 정의하지만 그 함수가 스크립트 안에서 한 번만 쓰인다면 분명 귀찮을 것이다.
- Bash 셸은 함수에 대한 라이브러리 파일을 만들어서 같은 라이브러리 파일을 원하는 어떤 스크립트에서나 쓸 수 있도록 기능을 제공한다. 이를 위한 첫번째 일은 스크립트에 필요한 기능들을 포함하는 공용 라이브러리 파일을 만드는 것이다. 다음은 세가지 간단한 함수를 정의하는 myfuncs 라는 간단한 라이브러리 파일이다.

```
#!/bin/bash

function addem {
    echo $[ $1 + $2 ]
}

function multem {
    echo $[ $1 * $2 ]
}

function divem {
    if [ $2 -ne 0 ]
    then
        echo $[ $1 / $2 ]
    else
        echo -1
    fi
}
[root@srv1 test]#
```

## 라이브러리 만들기 (Cont.)

- 이 예는 myfuncs 라이브러리 파일이 쉘 스크립트와 같은 디렉토리에 있는 것으로 가정한다. 다음은 myfuncs 라이브러리 파일을 사용하는 스크립트를 작성하는 예 이다

```
[root@srv1 test]# cat test14
#!/bin/bash

. ./myfuncs

value1=10
value2=5
result1=$(addem $value1 $value2)
result2=$(multem $value1 $value2)
result3=$(divem $value1 $value2)

echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
[root@srv1 test]#
[root@srv1 test]# ./test14
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
[root@srv1 test]#
```

## 텍스트 메뉴 만들기

- 교육내용
  - 텍스트 메뉴 만들기
  - 텍스트 위젯 구축하기
  - X 윈도우 그래픽 추가하기

## 텍스트 메뉴 만들기 (Cont.)

- 쉘 스크립트 메뉴의 핵심은 case 명령이다. case 명령은 사용자가 메뉴에서 어떤 문자를 선택했는가에 따라 특정한 명령을 수행한다.
- 메뉴 레이아웃 만들기

```
clear
echo
echo -e "\t\t\tSys Admin Menu\n"
echo -e "\t1. Display disk space"
echo -e "\t2. Display logged on users"
echo -e "\t3. Display memory usage"
echo -e "\t0. Exit menu\n\n"
echo -en "\t\tEnter option: "
```

- 마지막 줄의 -en 옵션은 마지막에 줄 바꿈 문자를 추가하지 않고 줄을 표시한다. 이렇게 하면 커서가 사용자의 입력을 기다리는 줄의 끝에 남아 있기 때문에 메뉴가 좀 더 완성도를 갖추게 된다.

메뉴를 만드는 마지막 과정은 사용자의 입력을 받는 것이다. 이는 read 명령을 사용하여 수행된다. 한 글자만 입력을 받을 것이므로 read 명령에 딱 한 글자만 입력을 받는 -n 옵션을 사용한다. 이렇게 하면 사용자는 <Enter> 키를 누를 필요 없이 번호만 입력하면 된다.

## 텍스트 메뉴 만들기 (Cont.)

- 메뉴 함수 만들기

- 쉘 스크립트 메뉴 옵션은 개별 함수를 묶은 그룹으로서 만드는 게 더 쉽다. 이렇게 하면 단순하면서도 간결한, 쉽게 따라할 수 있는 case 명령을 만들 수 있다. 이를 위해서는 각각의 메뉴 옵션마다 개별 함수를 만들어야 한다. 메뉴의 쉘 스크립트를 만드는 첫 번째 단계는 스크립트가 어떤 코드를 수행하게 할지를 결정하고 코드 안에서 개별 함수로 배치하는 것이다.
- 쉘 스크립트 메뉴에서 도움이 되는 한 가지 기능은 메뉴 레이아웃 자체를 함수로 만드는 것이다

```
function menu {
    clear
    echo
    echo -e "\t\t\tSys Admin Menu\n"
    echo -e "\t1. Display disk space"
    echo -e "\t2. Display logged on users"
    echo -e "\t3. Display memory usage"
    echo -e "\t0. Exit menu\n\n"
    echo -en "\t\tEnter option: "
    read -n 1 option
}
```

- menu 함수를 호출하면 쉽게 다시 메뉴를 표시할 수 있다

## 텍스트 메뉴 만들기 (Cont.)

### • 메뉴 로직 추가하기

- 이제 메뉴의 레이아웃과 함수를 만들었으니, 두 가지를 묶어서 프로그래밍 로직을 만들어야 한다. 앞서 언급한 것처럼 case 문이 필요하다
- case 명령은 메뉴에서 선택되는 글자에 따라서 적절한 함수를 불러야 한다. 잘못된 메뉴 항목 입력을 처리하기 위해서 기본값 case 명령 문자(별표)를 사용하는 것이 좋다

```
menu
case $option in
0)
break ;;
1)
diskspace ;;
2)
whoseon ;;
3)
memusage ;;
#
clear
echo "Sorry, wrong selection";;
esac
```

- 위 코드는 먼저 모니터 화면을 지우고 메뉴를 표시할 수 있는 menu 함수를 사용한다

## 텍스트 메뉴 만들기 (Cont.)

### • 모두 묶기

- 이제 모든 요소들을 한데 묶어 보고 상호작용이 어떻게 일어나는지 살펴보자

```
#!/bin/bash

function diskpace {
clear
df -k
}

function whoseon {
clear
who
}

function memusage {
clear
cat /proc/meminfo
}

function menu {
clear
echo
echo -e "\t\t\tSys Admin Menu\n"
echo -e "\t1. Display disk space"
echo -e "\t2. Display logged on users"
echo -e "\t3. Display memory usage"
echo -e "\t0. Exit menu\n\n"
echo -en "\t\tEnter option: "
read -n 1 option
}

while [ 1 ]
do
menu
case $option in
0)
break ;;
1)
diskpace ;;
2)
whoseon ;;
3)
memusage ;;
*)
clear
echo "Sorry, wrong selection";;
esac
echo -en "\n\n\t\t\tHit any key to continue"
read -n 1 line
done
clear
```



## 텍스트 메뉴 만들기 (Cont.)

- select 명령 사용하기

- 텍스트 메뉴를 만들 때 들이는 노력의 반은 메뉴 레이아웃을 만들고 사용자가 입력한 답을 얻는 데 들어간다는 것을 알 수 있다. Bash 셸은 자동으로 이러한 모든 작업을 수행하는 쉽고 간단한 유틸리티를 제공한다.
- Select 명령을 사용하면 커맨드라인 한 줄로 메뉴를 만든 다음 답을 입력 받아 자동으로 처리할 수 있다. 형식은 다음과 같다

```
select variable in list
do
    commands
done
```

- list 매개변수는 메뉴를 만들기 위한 텍스트 항목의 목록으로 빈 칸으로 분리된다. select 명령은 각 명령을 번호 옵션과 함께 표시한 다음 메뉴 선택을 위해 PS3 의 환경 변수가 정의하는 특별한 프롬프트를 표시한다

## 텍스트 메뉴 만들기 (Cont.)

- select 명령 사용한 예

```
#!/bin/bash

function diskspace {
    clear
    df -k
}

function whoseon {
    clear
    who
}

function memusage {
    clear
    cat /proc/meminfo
}

PS3="Enter option: "
select option in "Display disk space" "Display logged on users" "Display memory usage" "
Exit program"
do
    case $option in
        "Exit program")
            break ;;
        "Display disk space")
            diskspace ;;
        "Display logged on users")
            whoseon ;;
        "Display memory usage")
            memusage ;;
        *)
            clear
            echo "Sorry, wrong selection";;
    esac
done
clear
```

## 창 만들기

- dialog 패키지

- ANSI 이스케이프 제어 코드를 이용하여 텍스트 환경에서 표준 창 대화상자를 재현한다. 이러한 대화상자를 손쉽게 셸 스크립트에 통합해서 사용자와 상호작용하는 스크립트를 만들 수 있다.

- 설치
  - yum install dialog [CentOS]
  - apt-get install dialog [Ubuntu]

- dialog 패키지가 지원하는 위젯

위젯	설명	위젯	설명
calendar	날짜를 선택하는 달력제공	pause	지정한 일시 정지 상태 시간을 보여주는 계량기 표시
checklist	각 항목을 켜거나 끌 수 있는 여러 항목 표시	passwordbox	입력받은 텍스트를 숨기는 단일 텍스트 상자를 표시
form	레이블과 입력을 위한 텍스트 필드로 구성된 양식 생성	passwordform	레이블과 숨겨진 텍스트 필드로 구성된 양식을 표시
fselect	파일을 검색하는 파일 선택 창을 제공	radiolist	하나의 아이템만을 선택할 수 있는 메뉴항목 그룹 제공
gauge	진행 비율을 나타내는 계량기를 표시	tailbox tail	명령을 사용하여 스크롤 창에서 파일의 텍스트를 표시
infobox	응답을 기다리지 않고 메시지 표시	tailboxbg	tailbox 와 동일하지만 백그라운드 모드에서 작동
inputbox	텍스트 입력을 위한 단일 텍스트 양식 상자 표시	textbox	스크롤 할 수 있는 창에 파일의 내용을 표시
inputmenu	편집메뉴 제공	timebox	시, 분, 초 선택 창을 제공
menu	선택할 수 있는 목록을 표시	yesno	Yes 와 No 버튼이 있는 간단한 메시지를 제공
msgbox	메시지를 표시하고 사용자가 OK 버튼을 누르도록 요구		

## 창 만들기 (Cont.)

- msgbox 위젯

- 창에 간단한 메시지를 표시하고 사용자의 OK 버튼 클릭을 기다린다
- 일반적인 표시 형식은 다음과 같다

dialog --msgbox text height width

예) dialog --title Testing --msgbox "This is a test" 10 20



## 창 만들기 (Cont.)

- yesno 위젯

예) `dialog --title "Please answer" --yesno "Is this thing on?" 10 20`



## 창 만들기 (Cont.)

- inputbox 위젯

- 사용자가 텍스트 문자열을 입력할 수 있는 간단한 텍스트 상자 영역을 제공한다. dialog 명령은 STDERR 에 텍스트 문자열의 값을 전송한다. 답을 얻기 위해서는 STDERR 을 리다이렉트 해야 한다

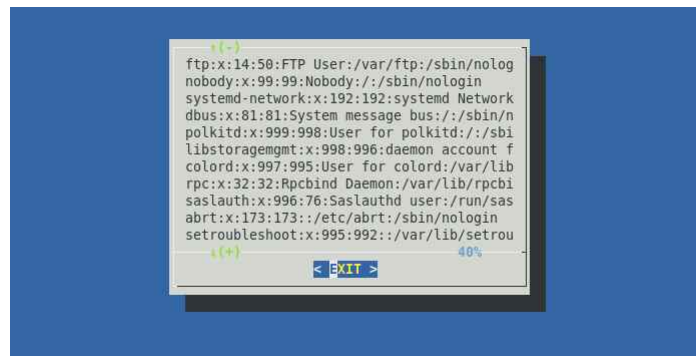
예) `dialog --title "Your age" --inputbox "Enter your age:" 10 20 2>age.txt`



## 창 만들기 (Cont.)

- textbox 위젯
  - 창에 많은 정보를 표시할 수 있는 좋은 방법이며 이 위젯은 매개변수로 지정된 파일에서 얻은 텍스트를 포함하는 스크롤 가능한 창을 만든다

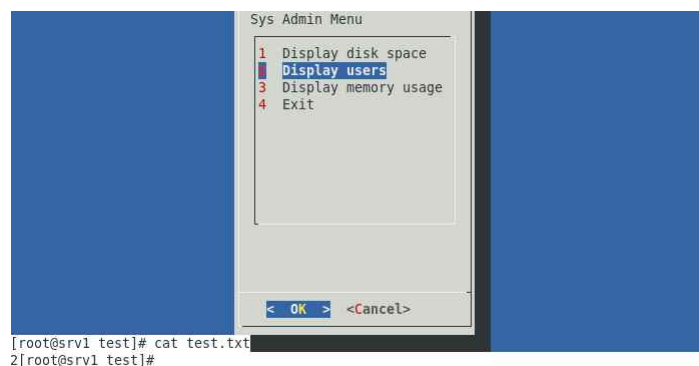
예) dialog --textbox /etc/passwd 15 45



## 창 만들기 (Cont.)

- menu 위젯
  - menu 위젯은 앞서 만든 텍스트 메뉴의 윈도우 버전을 만들 수 있다. 선택 태그와 텍스트만 지정해 주면 된다

예) dialog --menu "Sys Admin Menu" 20 30 1 "Display disk space" 2 "Display users" 3 "Display memory usage" 4 "Exit" 2> test.txt

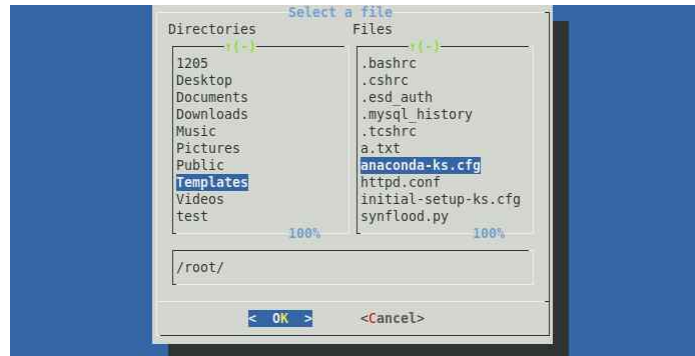


## 창 만들기 (Cont.)

- fselect 위젯

- 사용자가 파일 이름을 키보드로 입력하는 대신 fselect 위젯을 사용하면 파일 위치를 찾아 파일을 선택할 수 있다.

예) `dialog --title "Select a file" --fselect $HOME/ 10 50 2>file.txt`



## 창 만들기 (Cont.)

- dialog 옵션

위젯	설명	위젯	설명
--add-widget	<Esc> 또는 <Cancel> 버튼을 누르지 않으면 다음 대화상자로 넘어간다	--ignore	대화상자가 인식하지 못하는 옵션은 무시
--aspect	윈도우의 너비/높이 비율을 지정한다	--input-fd fd	STDIN 이외의 다른 파일 디스크립터를 지정한다
--backtitle	title 화면의 위쪽 끝에 배경으로 표시할 제목을 지정	--insecure	패스워드 위젯이 입력 때 별표 표시
--begin XY	윈도우의 왼쪽 상단이 시작되는 위치를 지정한다	--item-help	화면의 가장 아래에 checklist, radiolist, menu의 각 태그에 대한 도움말 열을 추가
--cancel-label	label Cancel 버튼의 대체 레이블을 지정한다	--keep-window	화면에서 이전 위젯을 지우지 않는다
--clear	기본 대화상자 배경색을 사용하여 디스플레이를 지운다	--max-input size	입력문자열의 최대 크기를 지정한다. 기본값은 2048
--colors	대화상자 텍스트에 ANSI 컬러 코드를 넣는다	--nocancel	Cancel 버튼 표시하지 않는다
--cr-wrap	대화상자 텍스트에서 줄바꿈 문자를 허용하고 강제로 줄 바꿈을 한다	--no-collapse	대화상자의 텍스트에서 탭을 빈칸으로 변환하지 않는다
--create-rc file	샘플 구성 파일을 지정된 파일로 덮어쓴다	--no-kill	모든 tailboxbg 대화상자를 백그라운드로 두고 프로세스의 SIGHUP을 비활성화시킨다
--defaultno	예/아니오 대화상자의 기본 버튼을 <No>로 만든다	--no-label label	<No> 버튼이 대체 레이블을 지정한다
--default-item string	checklist, form, menu 위젯의 기본값 항목을 정한다	--no-shadow	대화상자 윈도우에 그림자를 표시하지 않는다
--exit-label label	<Exit> 버튼의 대체 레이블을 지정한다	--ok-label	<OK> 버튼의 대체 레이블을 지정한다
--extra-button	<OK>와 <Cancel> 버튼 사이에 <Extra> 버튼을 표시한다	--output-fd fd	STDERR 이외의 다른 출력 파일 디스크립터를 지정한다
--extra-label label	<Extra> 버튼의 대체 레이블을 지정한다	--print-maxsize	출력을 위해 허용되는 대화상자 윈도우의 최대 크기 표시
--help	dialog 명령의 도움말 메시지를 표시한다	--print-size	각 다이얼로그 윈도우의 크기를 표시
--help-button	<OK>와 <Cancel> 버튼 사이에 <Help> 버튼을 표시한다	--print-version	dialog 버전을 표시
--help-label label	<Help> 버튼의 대체 레이블을 지정한다	--separate-output checklist	위젯의 결과를 한 줄에 인공부호 없이 표시한다
--help-status	<Help> 버튼을 눌렀을 때 보여주는 도움말 정보 다음에 checklist, radiolist, form 정보를 표시		

## 창 만들기 (Cont.)

- dialog 옵션

위젯	설명	위젯	설명
--separator string	각 위젯의 출력을 구분하는 문자열을 지정한다	--tab-len n	탭 문자가 차지하는 공간의 수를 지정하며 기본값은 8이다
--separate-widget string	각 위젯의 출력을 구분하는 문자열을 지정한다	--timeout sec	사용자가 아무런 입력도 하지 않을 때 오류 코드를 내고 종료할 때까지의 시간(초)을 지정한다
--shadow	각 윈도우의 오른쪽 아래에 그림자를 그린다	--title title	대화상자의 제목을 지정한다
--single-quoted	checkboxlist 출력이 필요한 경우 홑따옴표를 사용한다	--trim	대화상자 텍스트의 앞에 있는 빈 칸과 줄 바꿈을 없앤다
--sleep sec	대화상자 창을 처리한 후 지정된 시간(초) 동안 지연시킨다	--visit-items	항목의 목록을 포함하는 대화상자 윈도우에서 탭으로 선택되는 순서를 바꾼다
--stderr	출력을 STDERR 로 출력을 보내며 기본값이다	--yes-label label	Yes 버튼의 대체 레이블을 지정한다
--stdout	STDOUT 으로 출력을 보낸다		
--tab-correct	탭을 빈칸으로 변환한다		

## 창 만들기 (Cont.)

- 스크립트에서 dialog 명령 사용하기
  - 사용할 수 있는 Cancel 또는 No 버튼이 있다면 dialog 명령의 종료 상태 값을 확인하기
  - 결과값을 얻기 위해서는 STDERR 를 리다이렉트하기

```
#!/bin/bash

temp=$(mktemp -t test.XXXXXX)
temp2=$(mktemp -t test2.XXXXXXX)

function diskpace {
    df -k > $temp
    dialog --textbox $temp 20 60
}

function whoseon {
    who > $temp
    dialog --textbox $temp 20 50
}

function memusage {
    cat /proc/meminfo > $temp
    dialog --textbox $temp 20 50
}

while [ 1 ]
do
    dialog --menu "Sys Admin Menu" 20 30 10 1 "Display disk space" 2 "Display logged on users" 3 "Display memory usage" 0 "Exit" 2> $temp2

    if [ $? -eq 1 ]
    then
        break
    fi

    selection=$(cat $temp2)
```

```
case $selection in
1)
    diskpace ;;
2)
    whoseon ;;
3)
    memusage ;;
0)
    break ;;
*)
    dialog --msgbox "Sorry, invalid selection" 10 30
esac
done
rm -f $temp 2> /dev/null
rm -f $temp2 2> /dev/null
```

메뉴 대화창을 표시하는 무한 루프를 만들기 위해서 항상 true 값인 while 루프를 사용한다. 이는 어떤 함수가 호출되든 그 다음에는 메뉴 표시로 돌아온다는 것을 뜻한다.

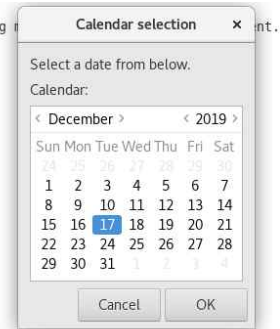
dialog 명령의 데이터를 저장할 두 개의 임시 파일을 만들기 위해 mktemp 명령을 사용한다. 먼저 \$temp는 df, whoseon, meminfo 명령의 출력을 저장하는 데 쓰이며 따라서 textbox 대화상자 안에 그 내용이 표시될 수 있다. 두 번째 임시파일인 \$temp2 는 메인 메뉴 대화상자에서 선택된 값을 저장하기 위해 사용된다.

## 그래픽 사용하기

- GNOME
  - GNOME 그래픽 환경은 표준 윈도우를 만들 수 있는 gdialog, zenity 를 제공한다

옵션	설명
--calendar	완전한 한 달의 달력을 표시
--entry	텍스트 대화 입력상자를 표시
--error	오류 메시지 대화상자를 표시
--file-selection	전체 경로와 파일 이름을 표시하는 대화상자를 표시
--info	정보 대화상자를 표시
--list	체크리스트 또는 라디오 버튼 목록 대화상자를 표시
--notification	알림 아이콘 표시
--progress	진행 표시줄 대화 상자를 표시
--question	예/아니오 질문 대화 상자를 표시
--scale	스케일 대화상자를 표시
--text-info	텍스트를 포함하는 텍스트 상자를 표시
--warning	경고 대화상자를 표시

```
[root@srv1 test]# zenity --calendar
Gtk-Message: 12:15:59.512: GtkDialog mapped before the main window.
ged.
```



## 그래픽 사용하기 (Cont.)

- 스크립트에서 zenity 사용하기
  - GNOME 그래픽 환경은 표준 윈도우를 만들 수 있는 gdialog, zenity 를 제공한다

```
#!/bin/bash

temp=$(mktemp -t test.XXXXXX)
temp2=$(mktemp -t test2.XXXXXX)

function diskpace {
  df -k > $temp
  zenity --text-info --title "Disk space" --filename=$temp --width 750 --height 10
}

function whoseon {
  who > $temp
  zenity --text-info --title "Logged in users" --filename=$temp --width 500 --height 1
}

function memusage {
  cat /proc/meminfo > $temp
  zenity --text-info --title "Memory Usage" --filename=$temp --width 300 --height 500
}

while [ 1 ]
do
  zenity --list --radiolist --title "Sys Admin menu" --column "Select" --column "Menu Item" FA
  LSE "Display diskpace" FALSE "Display users" FALSE "Display memory usage" FALSE "Exit" > $t
  emp2

  if [ $? -eq 1 ]
  then
    break
  fi

  selection=$(cat $temp2)

  case $selection in
    "Display disk space")
      diskpace ;;
    "Display users")
      whoseon ;;
    "Display memory usage")
      memusage ;;
    "Exit")
      break ;;
    *)
      zenity --info "Sorry, invalid selection"
  esac
done

[root@srv1 test]#
```

## 리눅스 셸 프로그래밍

---

### 부록