Shell Programming

- Sameer

Agenda

- **Basic of shell script.**
  - Simple shell script for start up
  - Declaring variables
  - Passing argument to shell script.
  - importing variable list
  - Using array (bash)
- **Using control statement**
  - If then else
  - Do while
  - For loop.
- **function**
  - Declaring user defined function
  - Calling functions
  - Passing arguments to function.
- **Calling one shell script from other**

Types of shell

In UNIX there are two major types of shells:

- The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
- The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows:

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow:

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

Shell Script - Concept

The basic concept of a shell script

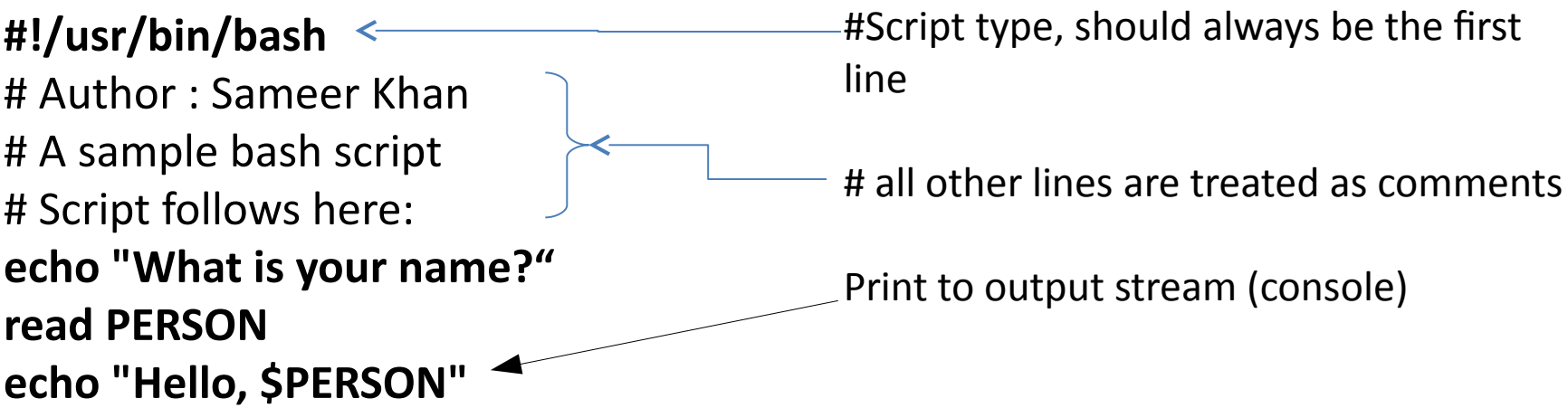
- A ***list of commands***, which are listed in the order of execution.
- ***Interpreted***. This means they are not compiled.

A good shell script will have comments (preceded by a pound sign, #) describing the steps.

There are ***conditional tests, loops, files*** to read and store data, and ***variables*** to read and store data, and the script may include ***functions***.

Shell Script

Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct. For example:



Make sure you give execute permission to the script for execution

```
chmod +x sample.sh
```

Variables

Variable Names:

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

Defining Variables:

- Variables are defined as follows::
- variable_name=variable_value
- For example: NAME=“Motorola"

Read-only Variables:

- The shell provides a way to mark variables as read-only by using the readonly command. After a variable is marked read-only, its value cannot be changed.
- For example, following script would give error while trying to change the value of NAME:

```
#!/bin/sh
NAME=“Motorola“
readonly NAME
NAME=“Sameer"
```

Unsetting Variables:

- Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.
- Following is the syntax to unset a defined variable using the unset command:
- **unset variable_name**

Variable Types

- Local Variables: A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell.
- Environment Variables: An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.

Special variables

For handling arguments passed to the script

Variable	Description
$0	The filename of the current script.
$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The number of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

Write a script to accept atleast 2 command line arguments, and print them.

Array Variables – available in **bash** but **NOT in sh**

Syntax of array initialization

- array_name[index]=value
- In **ksh** shell : *set -A array_name value1 value2 ... ValueN*
- In **bash** shell : *array_name=(value1 ... valueN)*

Accessing Array Values:

- ${array_name[index]}
- You can access all the items in an array in one of the following ways:
- ${array_name[*]}
- ${array_name[@]}

Operators

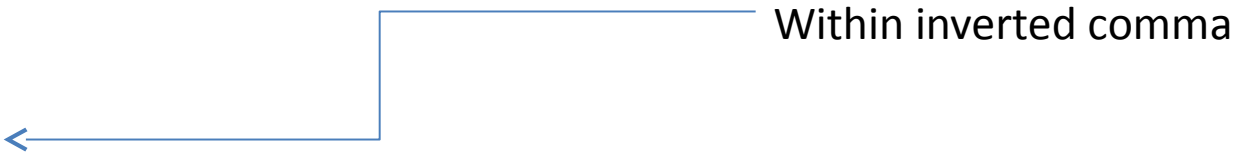
- Arithmetic Operators.
- Relational Operators.
- Boolean Operators.
- String Operators.
- File Test Operators.

Arithmatic Operators

Operator	Description	Example : let a="10" & B="20"
+	Addition - Adds values on either side of the operator	`expr $a + $b` will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
*	Multiplication - Multiplies values on either side of the operator	`expr $a * $b` will give 200
/	Division - Divides left hand operand by right hand operand	`expr $b / $a` will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
=	Assignment - Assign right operand in left operand	a=$b would assign value of b into a
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.

Available only in bash

Arithmetic Operator




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

- There ***must be spaces*** between operators and expressions for example *2+2 is not correct*, where as it should be written as 2 + 2.
- Complete expresssion should be enclosed between ``, called inverted commas.

The use of backticks (backquotes) in arithmetic expansion has been superseded by double parentheses -- ((...)) and $((...)) -- and also by the very convenient let construction.

$((EXPRESSION)) is arithmetic expansion.

```
z=$(($z+3))
z=$((z+3))
(( n += 1 ))                    # Increment.
(( $n += 1 ))  # is incorrect!

let z=z+3
let "z += 3"    #  Quotes permit the use of spaces in variable assignment.
                #  The 'let' operator actually performs arithmetic evaluation, rather than expansion.
```

Relational Operators - for ***numeric*** values

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	[$a -le $b] is true.

Relational Operators - for ***string*** values

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.	[-z $a] is not false.
Str	Check if str is not the empty string. If it is empty then it returns false.	[$a] is not false.

```
if [[ $string =~ .*My.* ]] then echo "It's there!" fi
if [[ "$string" == *My* ]] then echo "It's there!"; fi
if [ `echo $string || grep 'My' ` ] then; echo "It's there!"; fi;
case "$string" in *My*) # Do stuff ;; esac
if [ "$string" != "${string/My/}" ]; then echo "It's there!" fi
```

```
contains() {
   string="$1“
   substring="$2"
   if test "${string#*$substring}" != "$string"
   then
          return 0 # $substring is in $string
   else
       return 1 # $substring is not in $string
    fi
}
contains "abcd" "e" || echo "abcd does not contain e“
contains "abcd" "ab" && echo "abcd contains ab"
```

Boolean Operators

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true then condition would be true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

sh

```
if [ $a -gt $b -o $c -lt $b ]
```

Bash supports grouping of conditions. Notice that if cond uses "**[[**"

```
if [[ ( $a -gt $b ) || (( $c -lt $b ) && ($a -gt 5)) ]]
```

File Test Operators

Operator	Description
-b file	Checks if file is a block special file if yes then condition becomes true.
-c file	Checks if file is a character special file if yes then condition becomes true.
-d file	Check if file is a directory if yes then condition becomes true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.
-k file	Checks if file has its sticky bit set if yes then condition becomes true.
-p file	Checks if file is a named pipe if yes then condition becomes true.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.
-r file	Checks if file is readable if yes then condition becomes true.
-w file	Check if file is writable if yes then condition becomes true.
-x file	Check if file is execute if yes then condition becomes true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.
-e file	Check if file exists. Is true even if file is a directory but exists.

Conditional statements: if – elif – else – fi

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

Example

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

Presedence is –a then –o and last !

```
if [ $a -gt $b -o $c -lt $b ]
```

Conditional statements : case...esac

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

Similar to break statement in c

Example:

```
#!/bin/sh
FRUIT="kiwi“
case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
   *)  echo "`Unknown Fruit"
      exit 1 # Command to come out of the program with status 1
   ;;
esac
```

Loops

while condition
do
  Statement(s) to be executed if command is true
done

for var in word1 word2 ... wordN
do
  Statement(s) to be executed for every word.
done

until condition
do
  Statement(s) to be executed until command is true
done

select var in word1 word2 ... wordN
do
  Statement(s) to be executed.
done

Available in **ksh and bash**, NOT in sh. You can change the prompt displayed by the select loop by altering the variable PS3 using export

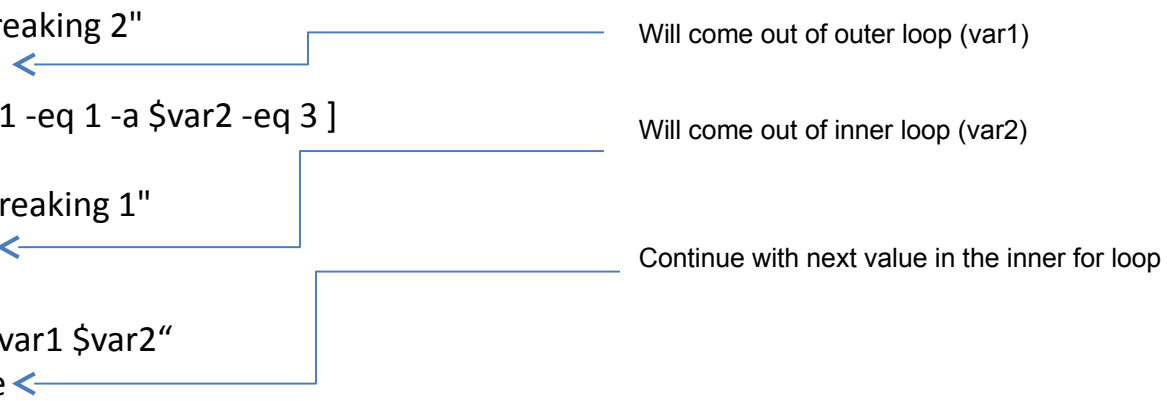
Note: All the loops support nesting concept which means, you can put one loop inside another similar or different loops

Loops – break & continue

- **break** statement is used to terminate the execution of the entire loop
- break command can also be used to exit from a nested loop using format **“break n”**
- **continue** and **“continue n”** statements are similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop to execute the next iteration of the loop.

- Example:
- #!/bin/sh

```
for var1 in 1 2 3
do
  for var2 in 0 3 5
  do
    if [ $var1 -eq 2 -a $var2 -eq 0 ]
    then
      echo "breaking 2"
       break 2
    elif [ $var1 -eq 1 -a $var2 -eq 3 ]
    then
       echo "breaking 1"
       break
    else
       echo "$var1 $var2“
      continue
    fi
  done
done
```



Metacharacters

- * ? [ ] ' " \ $; & ( ) | ^ < > new-line space tab

Quoting	Description
Single quote	All special characters between these quotes lose their special meaning.
Double quote	· Most special characters between these quotes lose their special meaning with these exceptions: $ ` \$ \' \" \\ All other \ characters are literal (not special).
Backslash	Any character immediately following the backslash loses its special meaning.
Back Quote	Anything in between back quotes would be treated as a command and would be executed.

Executing shell commands from script

```
#!/bin/sh

ls -ltr

DATE=`date`
echo "Date is $DATE“

USERS=`who | wc -l`
echo "Number of logged in users is $USERS"
```

Redirection

- “>” is used for redirecting the output to a file
  - Ex: echo line 1 > users.txt
  - “>” would overwrite on whatever was present in file users.txt
  - “>>” would append to end of file users.txt
- “<“ is used for reading input
  - Ex: wc -l < users.txt
- A *“<<”* is used to redirect input into an interactive shell script or program.

```
loginserv1:~/shell_learn > wc -l << STOP
> This is a simple example
> Of using here
> document
> stop
> STOP
      4
```

Reading line in for loop using redirection

```
while read line
    do
let COUNT=$COUNT+1
done < filename.txt
```

Function

- To declare a function, simply use the following syntax:

```
function_name () {
  list of commands
}
```

- **Passing parameters to a Function:**
- $1, $2 and so on represent parameters passed to a function.
- $# gives number of parameters received by a function
- **Returning value from a function**
- return statement is used for returning a numeric value back
- String values cannot be returned using return statement

- Shell support nested and recursive function calls

Function - example

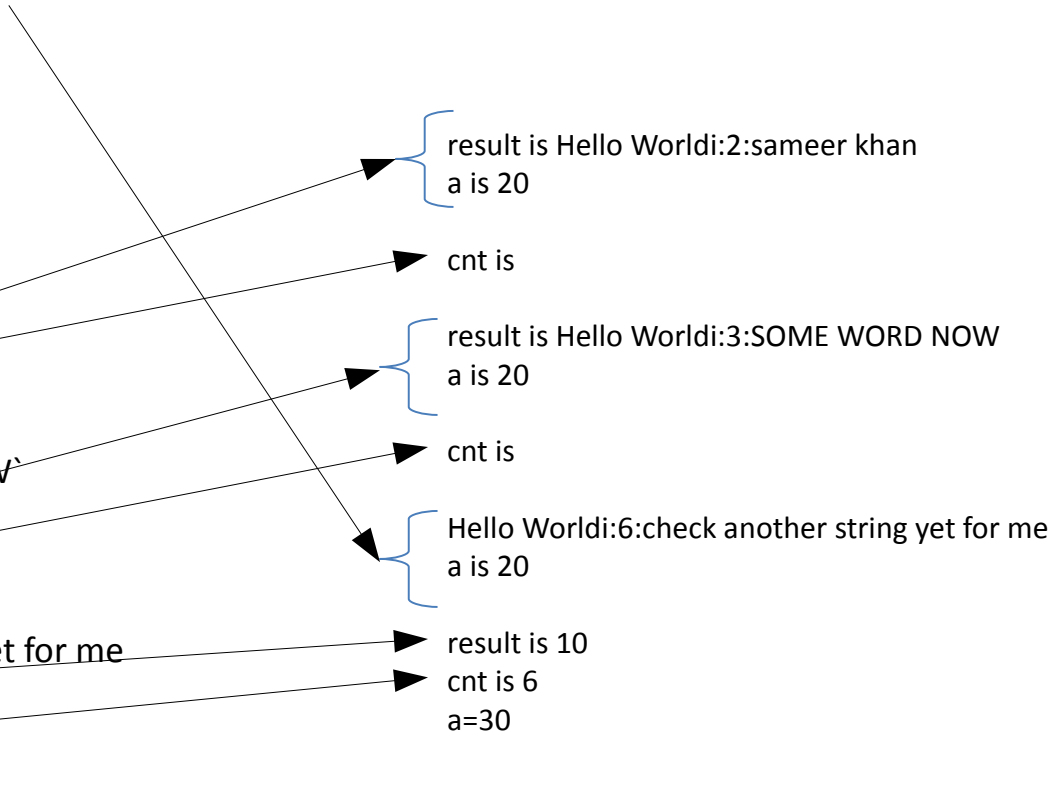
```
#!/bin/sh

a=“20”
# Define your function here
hello ()
{
  cnt=$#
  echo "Hello Worldi:$cnt:$@"
  echo “a is $a“
  a=“30”
  return 10
}

# Invoke your function
rtc=$(hello sameer khan)
echo "result is $rtc"
echo "cnt is $cnt“

rtc=`hello SOME WORD NOW`
echo "result is $rtc"
echo "cnt is $cnt"

hello check another string yet for me
echo "result is $?“
echo "cnt is $cnt"
echo "a=$a"
```



Scope of variable

Scope of variable is throughout the script – are always global to script session

A variable created in one function can be accessed in another function

A variable defined within “(“ and “)” has scope within that block

Ex:

```
(
 newVar="30"
 echo "newVar is $newVar"
)


echo "outside block newVar is $newVar"
```

Variable Substitution

Form	Description
${var}	Substitue the value of var.
${var:-word}	If var is null or unset, word is substituted for var. The value of var does not change.
${var:=word}	If var is null or unset, var is set to the value of word.
${var:?message}	If var is null or unset, message is printed to standard error. This checks that variables are set correctly.
${var:+word}	If var is set, word is substituted for var. The value of var does not change.

Indirect references

```
a=letter_of_alphabet   # Variable "a" holds the name of another variable.
letter_of_alphabet=z

# Direct reference.
echo "a = $a"          # a = letter_of_alphabet

# Indirect reference.
  eval a=\$$a
# --------------------------------------------------------------------
# The 'eval' forces an update of $a, sets it to the updated value of \$$a.
# So, we see why 'eval' so often shows up in indirect reference notation.
# --------------------------------------------------------------------
  echo "Now a = $a"    # Now a = z
```

Debugging shell script

you can use -v and -x option with sh or bash command to debug the shell script. General syntax is as follows:
sh -vx { shell-script-name }
where **-v** Print shell input lines as they are read.
-x After expanding each simple-command, bash displays the expanded value of PS4 system variable, followed by the command and its expanded arguments.

Example:

$ cat test.sh

```
#
# Script to show debug of shell
#
tot=`expr $1 + $2`
echo $tot
```

```
loginserv1:~/shell_learn > bash -x
testDebug.sh 2 3
++ expr 2 + 3
+ tot=5
+ echo sum is 5
sum is 5
```

```
loginserv1:~/shell_learn > bash -vx testDebug.sh 2 3
#!/usr/bin/sh
#
# Script to show debug of shell
#
tot=`expr $1 + $2`
expr $1 + $2
++ expr 2 + 3
+ tot=5
echo sum is $tot
+ echo sum is 5
sum is 5
```

- In case of bash use
  - echo -e “using new line \n”
- Other topics
  - Trap
    - trap {commands} {signal number list}
  - getopts
  - Shift
    - The shift command moves the current values stored in the positional parameters (command line args) to the left one position.
    - $1 = -f $2 = foo $3 = bar
    - and you executed the shift command the resulting positional parameters would be as follows:
    - $1 = foo $2 = bar
    - EX:
    - echo "Current command line args are: \$1=$1, \$2=$2, \$3=$3"
    - shift
    - echo "After shift command the args are: \$1=$1, \$2=$2, \$3=$3"
    - Output:
    - Current command line args are: $1=-f, $2=foo, $3=bar
    - After shift command the args are: $1=foo, $2=bar, $3=
    - You can also move the positional parameters over more than one place by specifying a number with the shift command. Ex: shift 2

Exercise

- Read two numbers from command line argument and print their difference and average separated by a $
- Read three values from user and print least of all
- Shell script to print given numbers sum of all digit
- Shell script to print contains of file from given line number to next given number of lines
- Sort the given five number in ascending order (use of array)
- Print nos. as 5,4,3,2,1 using while loop
- Print all prime numbers less than 100
- Write a function to return square of a given number

Exercise

- Shell script to determine whether given file exist or not
- Shell script to implements menu based system
  - Ex: ask user to select from tea or coffee and give the location of tea machine or cofee machine
- Get all subprocesses spawned by given process id through shell script
- Get OS information thru shell script
- Script to convert file names from UPPERCASE to lowercase file names or vice versa.
- Script to read all lines containing MOTOROLA and extracting all 7th column/field - Use awk in script
- Script which accept two words from user then search first word in given file and replace it with second word – use sed in script

BACKUP