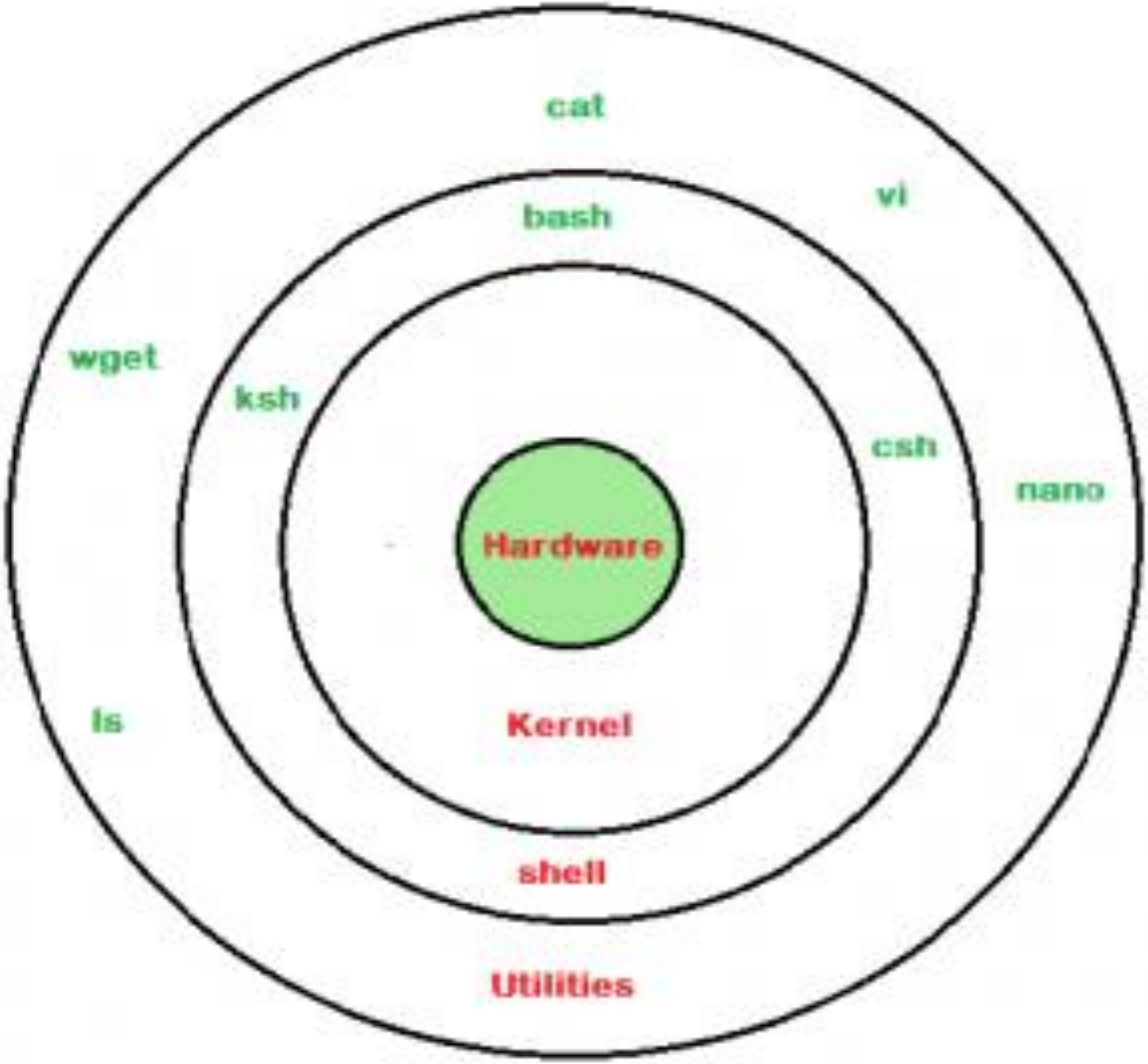
Shell Scripting

What is Shell

- A shell is special user program which provide an interface to user to use operating system services.
- Shell accept human readable commands from user and convert them into something which kernel can understand.
- It is a command language interpreter that execute commands read from input devices such as keyboards or from files.

What is Shell

- A shell is special user program which provide an interface to user to use operating system services.
- Shell accept human readable commands from user and convert them into something which kernel can understand.
- It is a command language interpreter that execute commands read from input devices such as keyboards or from files.


cat
vi
bash
wget
ksh
csh
nano
Hardware
ls
Kernel
shell
Utilities

TYPES OF SHELLS

There are four shells

- Bourne shell(sh),
- Korn shell(ksh),
- C shell(csh) and
- Bourne Again Shell (bash).

Basic Shell Programming

- A script is a file that contains shell commands
  - data structure: variables
  - control structure: sequence, decision, loop
- Shebang line for bash shell script:

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

- to run:
  - make executable: `% chmod +x script`
  - invoke via: `% ./script`

Bash shell programming

- Input
  - prompting user
  - command line arguments
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select
- Functions
- Traps

VARIABLE

- A variable is a character string to which we assign a value.
- The value assigned could be a number, text, filename, device, or any other type of dat
- Valid variables
- _abc
- Ab_c
- Ab_1
- Invalid Variables
- 1_ab
- -ab
- Ab-cd
- Ab_c!

Special shell variables

Parameter	Meaning
$0	Name of the current shell script
$1-$9	Positional parameters 1 through 9
$#	The number of positional parameters
$*	All positional parameters, “$*” is one string
$@	All positional parameters, “$@” is a set of strings
$?	Return status of most recently executed command
$$	Process id of current process

EXAMPLES: COMMAND LINE ARGUMENTS

```
% set tim bill ann fred
      $1  $2   $3  $4
% echo $*
tim bill ann fred
% echo $#
4
% echo $1
tim
% echo $3 $4
ann fred
```

The ‘set’ command can be used to assign values to positional parameters

ARRAY VARIABLE

- This can hold multiple values at the same time.
- Arrays provide a method of grouping a set of variables.
- syntax of array initialization
- Array=(va1 va2 va3)
- echo “first value=${Array[0]}”

OPERATORS

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

Arithmetic Operators

- shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**.
- C=`expr 1 + 1`
- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between ` `, called the backtick.

Arithmetic Operators

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b`
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b`
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b`
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a`
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a`
= (Assignment)	Assigns right operand in left operand	a = $b
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b]
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b]

- all the conditional expressions should be inside square braces with spaces around them,
- for example
- **[$a == $b]** is correct
- **[$a==$b]** is incorrect.

Relational Operators

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

Boolean Operators

x=5 y=10

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 the condition becomes true.	[$x -lt 10 -o $y -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$x -lt 20 -a $y -gt 100] is false.

String Operators

x=“ab” y=“fg”

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

User input

- shell allows to prompt for user input

Syntax:

```
read varname [more vars]
```

- or

```
read -p "prompt" varname [more vars]
```

- words entered by user are assigned to **`varname`** and "**`more vars`**"
- last variable gets rest of input line

User input example

```
#! /bin/bash
read -p "enter your name: " first last

echo "First name: $first"
echo "Last name: $last"
```

BASH CONTROL STRUCTURES

- if-then-else
- case
- loops
  - for
  - while
  - until
  - select

IF STATEMENT

```
if command
then
  statements
fi
```

- statements are executed only if **`command`** succeeds, i.e. has return status “0”

TEST COMMAND

Syntax:

```
  test expression
  [ expression ]
```

- evaluates ‘expression’ and returns true or false

Example:

```
read –p “Enter your password= “ pass
if test “$pass“ == “admin”
     then
     echo “Password Verfied"
  fi
```

The simple if statement

```
if [ condition ]; then
  statements
fi
```

- executes the statements only if **`condition`** is true

The if-then-else statement

```
if [ condition ]; then
   statements-1
else
  statements-2
fi
```

- executes statements-1 if condition is true
- executes statements-2 if condition is false

THE IF…STATEMENT

```
if [ condition ]; then
       statements
elif [ condition ]; then
        statement
else
        statements
fi
```

- The word **elif** stands for "else if"
- It is part of the if statement and cannot be used by itself

Relational Operators

Meaning	Numeric	String
Greater than	-gt	
Greater than or equal	-ge	
Less than	-lt	
Less than or equal	-le	
Equal	-eg	= or ==
Not equal	-ne	!=
str1 is less than str2		str1 < str2
str1 is greater str2		str1 > str2
String length is greater than zero		-n str
String length is zero		-z str

COMPOUND LOGICAL EXPRESSIONS

`!`	not
`&&`	and
`||`	or

and, or
must be enclosed within

[[ ]]

Example: Using the ! Operator

```
#!/bin/bash

read -p "Enter years of work: " Years
if [ ! "$Years" -lt 20 ]; then
   echo "You can retire now."
else
   echo "You need 20+ years to retire"
fi
```

EXAMPLE: USING THE && OPERATOR

```
#!/bin/bash

Bonus=500
read -p "Enter Status: " Status
read -p "Enter Shift: " Shift
if [[ "$Status" = "H" && "$Shift" = 3 ]]
then
   echo "shift $Shift gets \$$Bonus bonus"
else
   echo "only hourly workers in"
   echo "shift 3 get a bonus"
fi
```

Example: Using the | | Operator

```
#!/bin/bash

read -p "Enter calls handled:" CHandle
read -p "Enter calls closed: " CClose
if [[ "$CHandle" -gt 150 || "$CClose" -gt 50 ]]
   then
   echo "You are entitled to a bonus"
else
   echo "You get a bonus if the calls"
   echo "handled exceeds 150 or"
   echo "calls closed exceeds 50"
fi
```

File Testing

	Meaning
-d file	True if ‘file’ is a directory
-f file	True if ‘file’ is an ord. file
-r file	True if ‘file’ is readable
-w file	True if ‘file’ is writable
-x file	True if ‘file’ is executable
-s file	True if length of ‘file’ is nonzero

EXAMPLE: FILE TESTING

```
#!/bin/bash
echo "Enter a filename: "
read file
if [ ! -r $file ]
 then
   echo "File is not read-able"

fi
```

Example: File Testing

```
#! /bin/bash

echo "Enter a filename: "
read file
if [[ ! -f $file || ! -r $file || ! -w $file ]]
then
  echo "File $file is not accessible"

fi
```

Example: if… Statement

```
# The following THREE if-conditions produce the same result

* DOUBLE SQUARE BRACKETS
read -p "Do you want to continue?" reply
if [[ $reply = "y" ]]; then
   echo "You entered " $reply
fi

* SINGLE SQUARE BRACKETS
read -p "Do you want to continue?" reply
if [ $reply = "y" ]; then
   echo "You entered " $reply
fi

* "TEST" COMMAND
read -p "Do you want to continue?" reply
if test $reply = "y"; then
   echo "You entered " $reply
fi
```

The case Statement

- use the case statement for a decision that is based on multiple choices

Syntax:

```
case word in
    pattern1) command-list1
    ;;
    pattern2) command-list2
    ;;
    patternN) command-listN
    ;;
esac
```

CASE PATTERN

- checked against word for match
- may also contain:

  `*`

  `?`

  `[ … ]`

  `[:class:]`
- multiple patterns can be listed via:

  `|`

Example 1: The case Statement

```
#!/bin/bash
echo "Enter Y to see all files including hidden
  files"
echo "Enter N to see all non-hidden files"
echo "Enter q to quit"

read -p "Enter your choice: " reply

case $reply in
  Y|YES) echo "Displaying all (really…) files"
         ls -a ;;
  N|NO)  echo "Display all non-hidden files..."
         ls ;;
  Q)     exit 0 ;;

  *) echo "Invalid choice!"; exit 1 ;;
esac
```

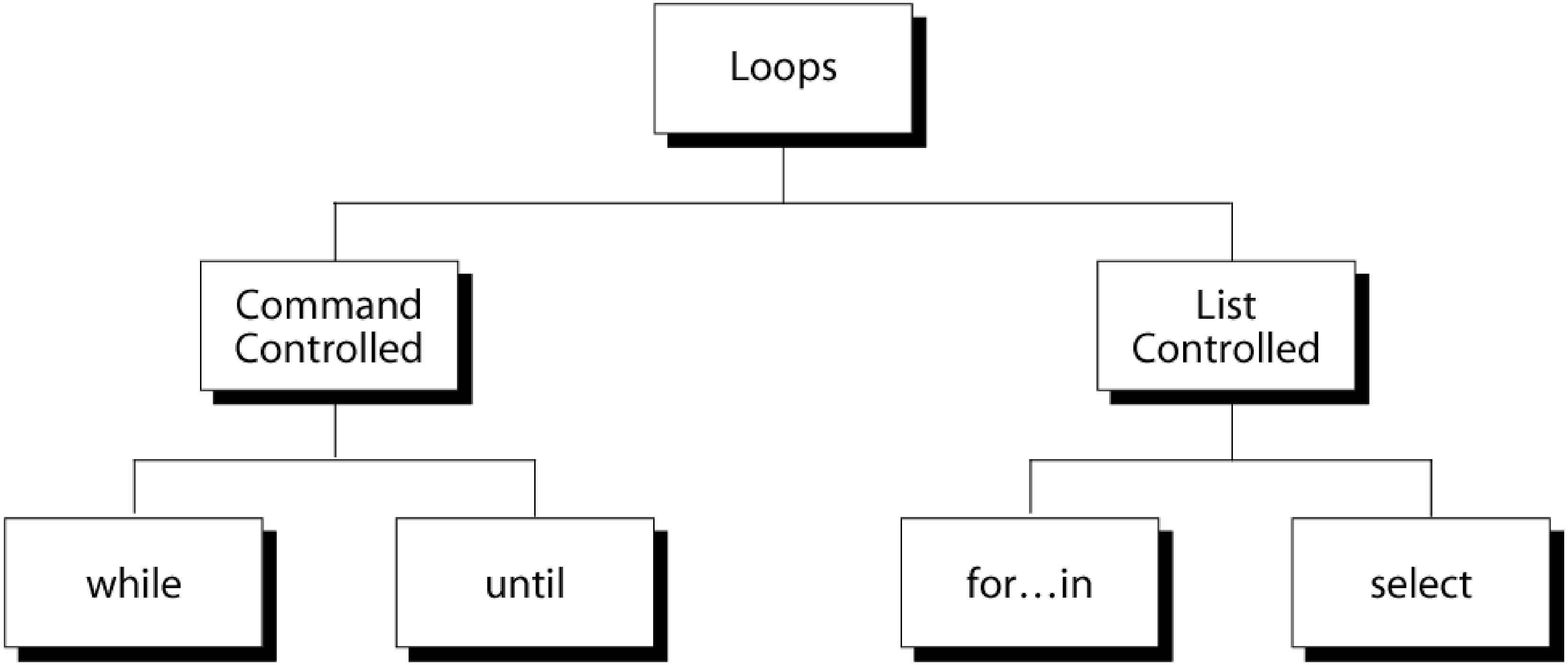
Bash programming: so far

- Data structure
  - Variables
  - Numeric variables
  - Arrays
- User input
- Control structures
  - if-then-else
  - case

BASH PROGRAMMING: STILL TO COME

- Control structures
  - Repetition
    - do-while, repeat-until
    - for
    - select
- Functions
- Trapping signals

Repetition Constructs



The while Loop

- Purpose:

  To execute commands in "command-list" as long as "expression" evaluates to true

Syntax:

```
while [ expression ]
do
    command-list
done
```

Example: Using the while Loop

```
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 10 ]
do
   echo The counter is $COUNTER
   let COUNTER=$COUNTER+1
done
```

Example: Using the while Loop

```
#!/bin/bash

Cont="Y"
while [ $Cont = "Y" ]; do
  ps -A
  read -p "want to continue? (Y/N)" reply
  Cont=`echo $reply | tr [:lower:] [:upper:]`
done
echo "done"
```

Example: Using the while Loop

```
#!/bin/bash

x=1
while [ $x -le 5 ]; do
 echo “Welcome $x times”
 x=$(( $x + 1 ))
done
```

The until Loop

- Purpose:

  To execute commands in “command-list” as long as “expression” evaluates to false

Syntax:

```
until [ expression ]
do
    command-list
done
```

Example: Using the until Loop

```
#!/bin/bash

COUNTER=20
until [ $COUNTER -lt 10 ]
do
   echo $COUNTER
   let COUNTER-=1
done
```

Example: Using the until Loop

```
#!/bin/bash

Stop="N"
until [ $Stop = "Y" ]; do
  ps -A
  read -p "want to stop? (Y/N)" reply
  Stop=`echo $reply | tr [:lower:] [:upper:]`
done
echo "done"
```

The for Loop

- Purpose:

  To execute commands as many times as the number of words in the "argument-list"

Syntax:

```
for variable in argument-list
do
    commands
done
```

Example 1: The for Loop

```
#!/bin/bash

for i in 7 9 2 3 4 5
do
   echo $i
done
```

Example 2: Using the for Loop

```
#!/bin/bash
# compute the average weekly
  temperature

for num in 1 2 3 4 5 6 7
do
   read -p "Enter temp for day $num: "
  Temp
   let TempTotal=$TempTotal+$Temp
done

let AvgTemp=$TempTotal/7
echo "Average temperature: " $AvgTemp
```

USING COMMA IN THE BASH C-STYLE FOR LOOP

```
#!/bin/bash

for  ((i=1, j=10;  i <= 5 ; i++, j=j+5))

do

echo “Number $i: $j”

done
```

SELECT COMMAND

- Constructs simple menu from word list
- Allows user to enter a number instead of a word
- User enters sequence number corresponding to the word

Syntax:

```
select WORD in LIST
do
    RESPECTIVE-COMMANDS
done
```

- Loops until end of input, i.e. ^d (or ^c)

Select example

```
#! /bin/bash
select var in alpha beta gamma
do
     echo $var
done
```

- Prints:

```
1) alpha
2) beta
3) gamma
#? 2
beta
#? 4
#? 1
alpha
```

Select detail

- PS3 is select sub-prompt
- $REPLY is user input (the number)

```
#! /bin/bash
PS3="select entry or ^D: "
select var in alpha beta
do
     echo "$REPLY = $var"
done
```

```
Output:
select ...
1) alpha
2) beta
? 2
2 = beta
? 1
1 = alpha
```

Select example

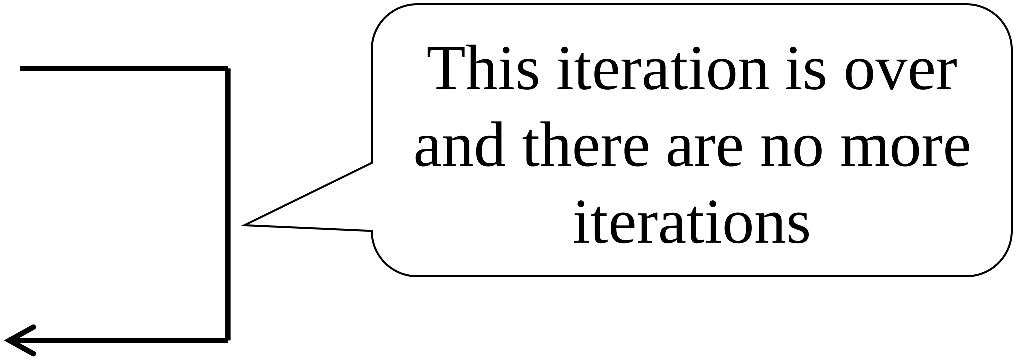
```
#!/bin/bash
echo "script to make files private"
echo "Select file to protect:"

select FILENAME in *
do
  echo "You picked $FILENAME ($REPLY)"
  chmod go-rwx "$FILENAME"
  echo "it is now private"
done
```

BREAK AND CONTINUE

- Interrupt for, while or until loop
- The break statement
  - transfer control to the statement AFTER the done statement
  - terminate execution of the loop
- The continue statement
  - transfer control to the statement TO the done statement
  - skip the test statements for the current iteration
  - continues execution of the loop

The break command

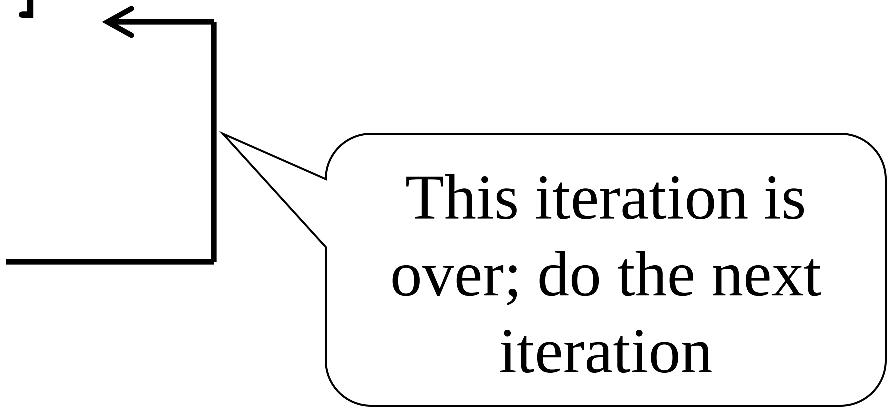
```
while [ condition ]
do
    cmd-1
    break
    cmd-n
done
echo "done"
```



This iteration is over and there are no more iterations

THE CONTINUE COMMAND

```
while [ condition ]
do
    cmd-1
    continue
    cmd-n
done
echo "done"
```



This iteration is over; do the next iteration

EXAMPLE:

```
for index in 1 2 3 4 5 6 7 8 9 10
do
        if [ $index -le 3 ]; then
             echo "continue"
             continue
        fi
        echo $index
        if  [ $index -ge 8 ]; then
             echo "break"
             break
        fi
done
```

Bash shell programming

- Sequence
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select

DONE !

- Functions
- Traps

still to come

Shell Functions

- A shell function is similar to a shell script
  - stores a series of commands for execution later
  - shell stores functions in memory
  - shell executes a shell function in the same shell that called it
- Where to define
  - In .profile
  - In your script
  - Or on the command line
- Remove a function
  - Use unset built-in

Shell Functions

- must be defined before they can be referenced
- usually placed at the beginning of the script

Syntax:

```
function-name () {
    statements
}
```

EXAMPLE: FUNCTION

```
#!/bin/bash

test () {
  # This is a simple function
  echo "This is a test function."
  echo "Now exiting test function."
}

# declaration must precede call:

test
```

EXAMPLE: FUNCTION

```
#!/bin/bash
fun () { # A somewhat more complex function.
  JUST_A_SECOND=1
  let i=0
  REPEATS=10
  echo "And now the fun really begins."
  while [ $i -lt $REPEATS ]
  do
      echo "-------FUNCTIONS are fun-------->"
      sleep $JUST_A_SECOND
      let i+=1
  done
}
fun
```

ARRAY VARIABLE

- This can hold multiple values at the same time.
- Arrays provide a method of grouping a set of variables.
- syntax of array initialization
- Array=(va1 va2 va3)
- echo “first value=${Array[0]}”

Syntax	Result
arr=()	Create an empty array
arr=(1 2 3)	Initialize array
${arr[2]}	Retrieve third element
${arr[@]}	Retrieve all elements
${!arr[@]}	Retrieve array indices
${#arr[@]}	Calculate array size
arr[0]=3	Overwrite 1st element
arr+=(4)	Append value(s)
str=$(ls)	Save ls output as a string
arr=($(ls))	Save ls output as an array of files
${arr[@]:s:n}	Retrieve n elements starting at index s

```
#!/bin/bash
 #Declare a string array

Array=("PHP"  "Java"  "C#"  "C++"  "VB.Net"  "Python"  "Perl")

# Print array values in  lines
echo "Print every element in new line"

for val1 in ${Array[*]}; do
     echo $val1
done

echo ""

# Print array values in one line
echo "Print all elements in a single line"
for val2 in "${Array[*]}"; do
    echo $val2
done
echo ""
```

```
#!/bin/bash
i=0
p=0
n=0
z=0
while [ $i -le 9 ]
do
read -p "Enter a Number: " a[$i]

if [ ${a[$i]} -gt 0 ]
then
    p=$(($p+1))
else
    if [ ${a[$i]} -eq 0 ]
    then
      z=$(($z+1))
    else
        n=$(($n+1))
    fi
fi
i=$(($i+1))
done
echo "+ve number= $p"
echo "-ve no= $n"
echo "zero= $z"
```

```
# !/bin/bash

clear
read -p "string= " str
echo
len=`echo $str | wc -c`
echo "len=$len"
len=$((len-1))
echo "len=$len"
i=1
j=`expr  $len / 2`
echo "j=$j"
while test $i -le $j
do
k=`echo $str | cut -c $i`
l=`echo $str | cut -c $len`
if test $k != $l
then
echo "String is not palindrome"
exit
fi
i=`expr $i + 1`
len=`expr $len - 1`
done
echo "String is palindrome"
```