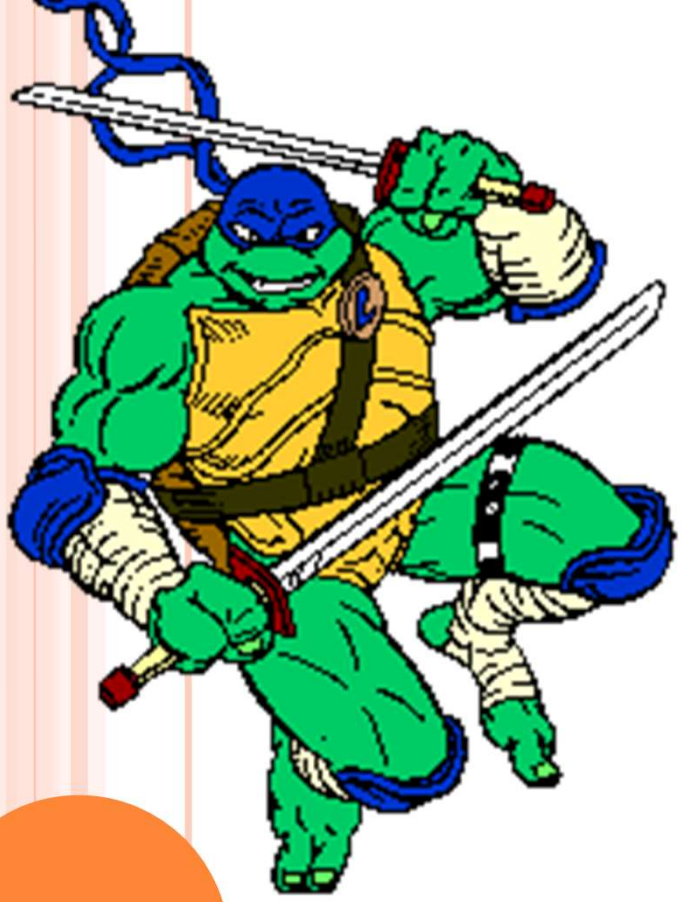
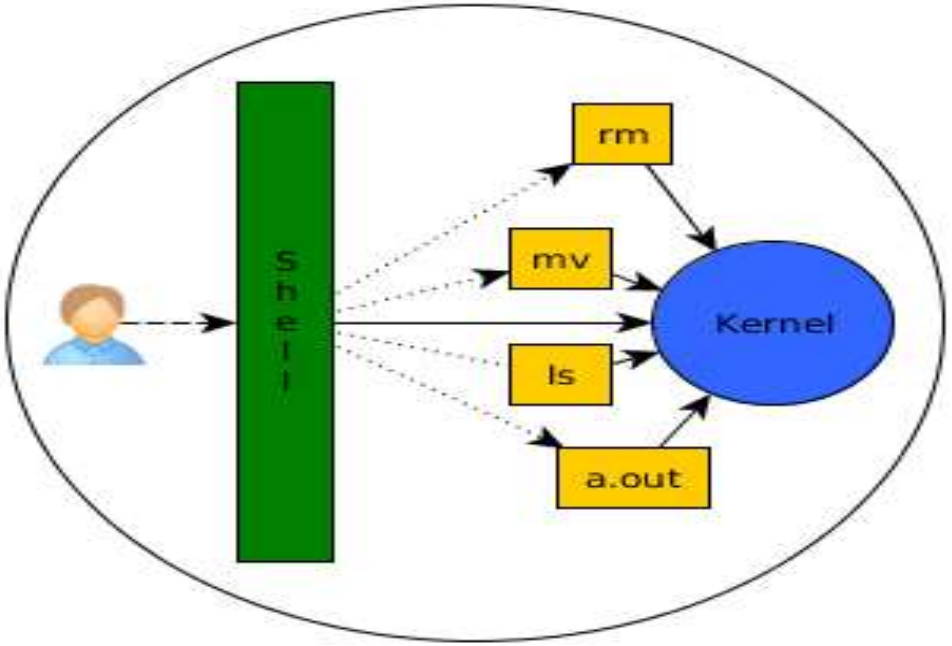


Shell Programming



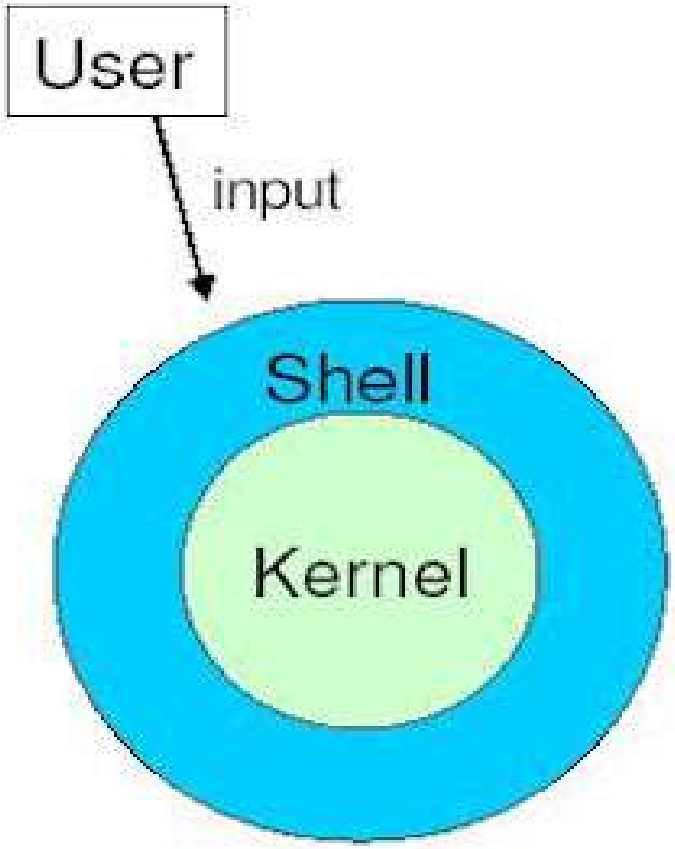
SHELLS

- A shell can be used in one of two ways:
  - A *command interpreter*, used interactively
  - A *programming language*, to write shell scripts (your own custom commands)

Types of shell

There are several different shells available for Unix. The most popular are described here.

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


User
input
Shell
Kernel

SHELL SCRIPTS

- A shell script is just a file containing shell commands, but with a few extras:
  - The first line of a shell script should be a comment of the following form:

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

    for a Bourne shell script. Bourne shell scripts are the most common, since C Shell scripts have buggy features.
  - A shell script must be readable and executable.

    ```
    chmod u+rx scriptname
    ```

  - As with any command, a shell script has to be "in your path" to be executed.
    - If "." is not in your PATH, you must specify "`./scriptname`" instead of just "`scriptname`"

SHELL SCRIPT EXAMPLE

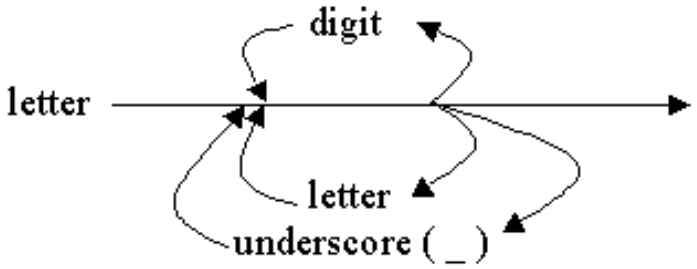
- Here is a "hello world" shell script:

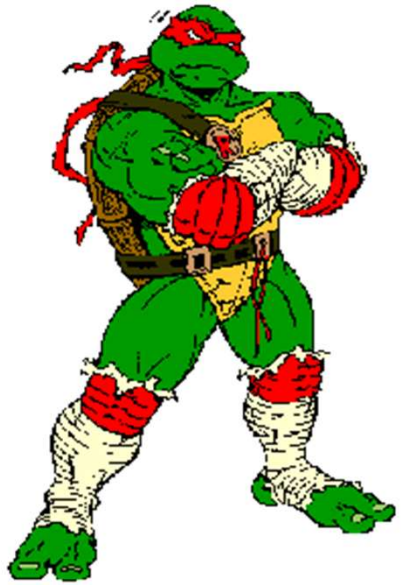
```
$ ls -l
-rwxr-xr-x  1 horner   48 Feb 19 11:50 hello*
$ cat hello
#!/bin/sh
# comment lines start with the # character
echo "Hello world"
$ hello
Hello world
$
```

  - The `echo` command functions like a print command in shell scripts.

SHELL VARIABLES

- The user variable name can be any sequence of letters, digits, and the underscore character, but the first character must be a letter.





- To assign a value to a variable:

```
number=25
name="Bill Gates"
```

- There cannot be any space before or after the “=“
- Internally, all values are stored as strings.

SHELL VARIABLES

- To use a variable, precede the name with a “$”:

```
$ cat test1
#!/bin/sh
number=25
name="Bill Gates"
echo "$number $name"
$ test1
25 Bill Gates
$
```

USER INPUT

- Use the `read` command to get and store input from the user.

```
$ cat test2
#!/bin/sh
echo "Enter name: "
read name
echo "How many girlfriends do you have? "
read number
echo "$name has $number girlfriends!"
$ test2
Enter name:
Bill Gates
How many girlfriends do you have?
too many
Bill Gates has too many girlfriends!
```

USER INPUT

- `read` reads one line of input from the keyboard and assigns it to one or more user-supplied variables.

```
$ cat test3
#!/bin/sh
echo "Enter name and how many girlfriends:"
read name number
echo "$name has $number girlfriends!"
$ test3
Enter name and how many girlfriends:
Bill Gates 63
Bill has Gates 63 girlfriends!
$ test3
Enter name and how many girlfriends:
BillG 63
BillG has 63 girlfriends!
$ test3
Enter name and how many girlfriends:
Bill
Bill has  girlfriends!
```

- Leftover input words are all assigned to the last variable.

$

- Use a backslash before $ if you really want to print the dollar sign:

```
$ cat test4
#!/bin/sh
echo "Enter amount: "
read cost
echo "The total is: \$$cost"
$ test4
Enter amount:
18.50
The total is $18.50
```

$

- You can also use single quotes for printing dollar signs.
- Single quotes turn off the special meaning of all enclosed dollar signs:

```
$ cat test5
#!/bin/sh
echo "Enter amount: "
read cost
echo ‘The total is: $’ "$cost"
$ test5
Enter amount:
18.50
The total is $ 18.50
```

EXPR

- Shell programming is not good at numerical computation, it is good at text processing.
- However, the `expr` command allows simple integer calculations.
- Here is an interactive Bourne shell example:

```
$ i=1
$ expr $i + 1
2
```

- To assign the result of an `expr` command to another shell variable, surround it with backquotes:

```
$ i=1
$ i=`expr $i + 1`
$ echo "$i"
2
```

EXPR

- The * character normally means “all the files in the current directory”, so you need a “\” to use it for multiplication:

```
$ i=2
$ i=`expr $i \* 3`
$ echo $i
6
```

- `expr` also allows you to group expressions, but the “(“ and “)” characters also need to be preceded by backslashes:

```
$ i=2
$ echo `expr 5 + \( $i \* 3 \)`
11
```

```
#addition of two numbers
echo –n “Enter the first number”
read a
read b
sum=`expr $a + $b`
echo “Sum of $a and $b is $sum”
```

EXPR EXAMPLE

```
$ cat test6
#!/bin/sh
echo "Enter height of rectangle: "
read height
echo "Enter width of rectangle: "
read width
area=`expr $height \* $width`
echo "The area of the rectangle is $area"
$ test6
Enter height of rectangle:
10
Enter width of rectangle:
5
The area of the ractangle is 50
$ test6
Enter height of rectangle:
10.1
Enter width of rectangle:
5.1
expr: non-numeric argument
```

Does not work for floats!

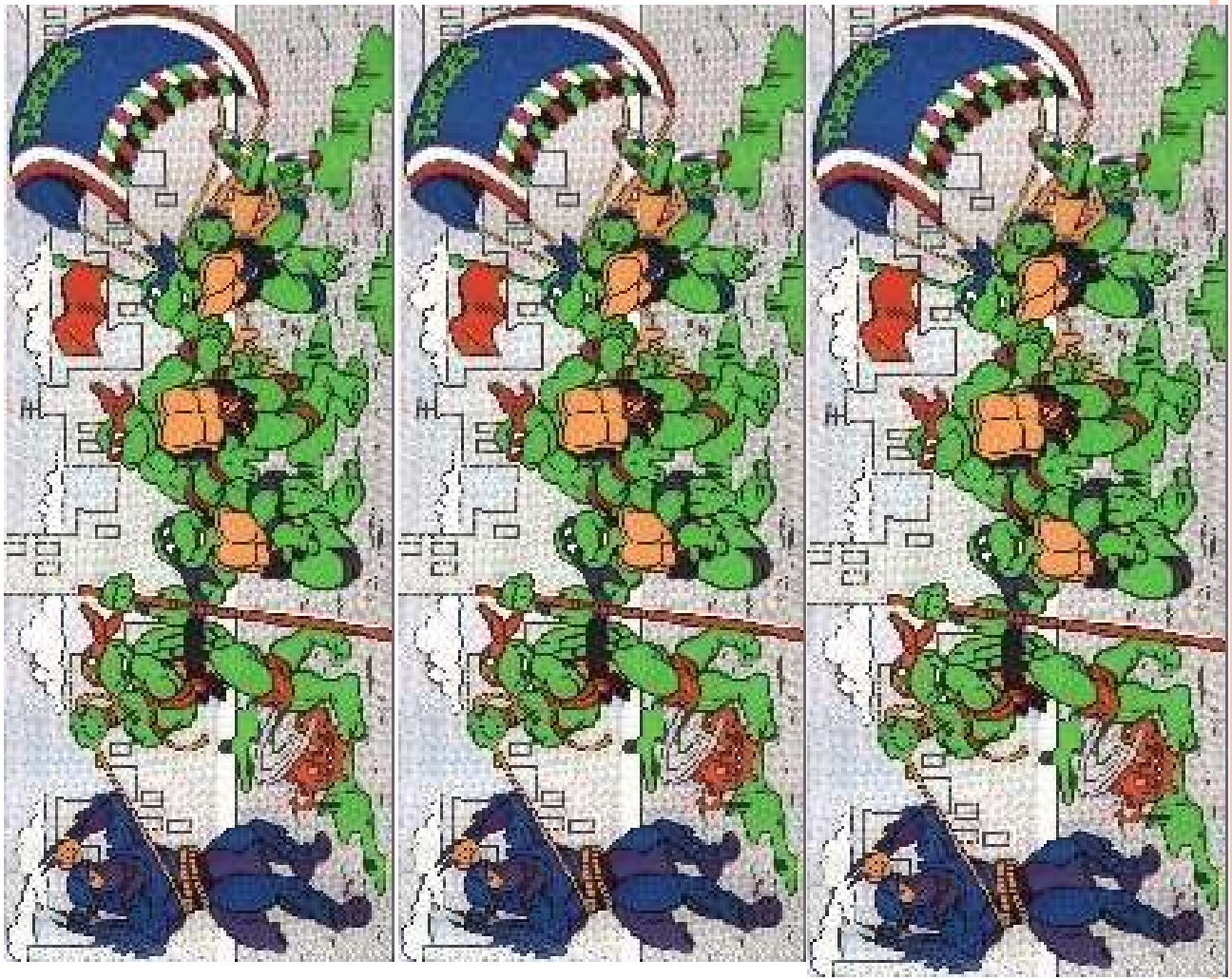
BACKQUOTES: COMMAND SUBSTITUTION

- A command or pipeline surrounded by backquotes causes the shell to:
  - Run the command/pipeline
  - Substitute the output of the command/pipeline for everything inside the quotes
- You can use backquotes anywhere:

```
$ whoami
gates
$ cat test7
#!/bin/sh
user=`whoami`
numusers=`who | wc -l`
echo "Hi $user! There are $numusers users logged on."
$ test7
Hi gates! There are       6 users logged on.
```

CONTROL FLOW

- The shell allows several control flow statements:
  - `if`
  - `while`
  - `for`



IF

- The `if` statement works mostly as expected:

```
$ whoami
clinton
$ cat test7
#!/bin/sh
user=`whoami`
if [ $user = "clinton" ]
then
        echo "Hi Bill!"
fi
$ test7
Hi Bill!
```

- However, the spaces before and after the square brackets [ ] are required.

IF THEN ELSE

- The `if then else` statement is similar:

```
$ cat test7
#!/bin/sh
user=`whoami`
if [ $user = "clinton" ]
then
        echo "Hi Bill!"
else
        echo "Hi $user!"
fi
$ test7
Hi horner!
```

IF ELIF ELSE

- You can also handle a list of cases:

```
$ cat test8
#!/bin/sh
users=`who | wc -l`
if [ $users -ge 4 ]
then
        echo "Heavy load"
elif [ $users -gt 1 ]
then
        echo "Medium load"
else
        echo "Just me!"
fi
$ test8
Heavy load!
```

BOOLEAN EXPRESSIONS

- Relational operators:

```
-eq, -ne, -gt, -ge, -lt, -le
```

- File operators:

```
-f file       True if file exists and is not a directory
-d file       True if file exists and is a directory
-s file       True if file exists and has a size > 0
```

- String operators:

```
-z string     True if the length of string is zero
-n string     True if the length of string is nonzero
s1 = s2       True if s1 and s2 are the same
s1 != s2      True if s1 and s2 are different
s1            True if s1 is not the null string
```

FILE OPERATOR EXAMPLE

```
$ cat test9
#!/bin/sh
if [ -f letter1 ]
then
        echo "We have found the evidence!"
        cat letter1
else
        echo "Keep looking!"
fi
$ test9
We have found the evidence!
How much would it cost to buy Apple Computer?
Best,
Bill
```

AND, OR, NOT

- You can combine and negate expressions with:

-a	And
-o	Or
!	Not

```
$ cat test10
#!/bin/sh
if [ `who | grep gates | wc -l` -ge 1 -a `whoami` != "gates" ]
then
     echo "Bill is loading down the machine!"
 else
     echo "All is well!"
 fi
$ test10
Bill is loading down the machine!
```

WHILE

- The `while` statement loops indefinitely, while the condition is true, such as a user-controlled condition:

```
$ cat test11
#!/bin/sh
resp="no"
while [ $resp != "yes" ]
do
        echo "Wakeup [yes/no]?"
        read resp
done
$ test11
Wakeup [yes/no]?
no
Wakeup [yes/no]?
y
Wakeup [yes/no]?
yes
$
```

WHILE

- `while` can also do normal incrementing loops:

```
$ cat fac
#!/bin/sh
echo "Enter number: "
read n
fac=1
i=1
while [ $i -le $n ]
do
        fac=`expr $fac \* $i`
        i=`expr $i + 1`
done
echo "The factorial of $n is $fac"
$ fac
Enter number:
5
The factorial of 5 is 120
```

BREAK

- The `break` command works like in C++, breaking out of the innermost loop :

```
$ cat test12
      #!/bin/sh
      while [ 1 ]
      do
            echo "Wakeup [yes/no]?"
            read resp
            if [ $resp = "yes" ]
            then
                  break
            fi
   done
   $ test12
   Wakeup [yes/no]?
   no
   Wakeup [yes/no]?
   y
   Wakeup [yes/no]?
   yes
   $
```

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

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/sh
read -p "enter your name: " first last

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

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

BASH CONTROL STRUCTURES

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

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:

```
if test -w "$1"
   then
   echo "file $1 is write-able"
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 [[ ]]

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
```

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

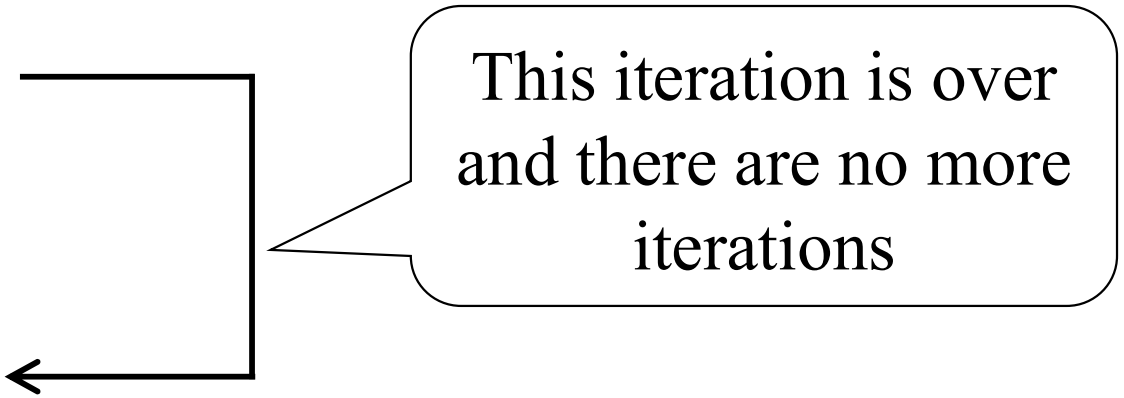
```
#!/bin/bash

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

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

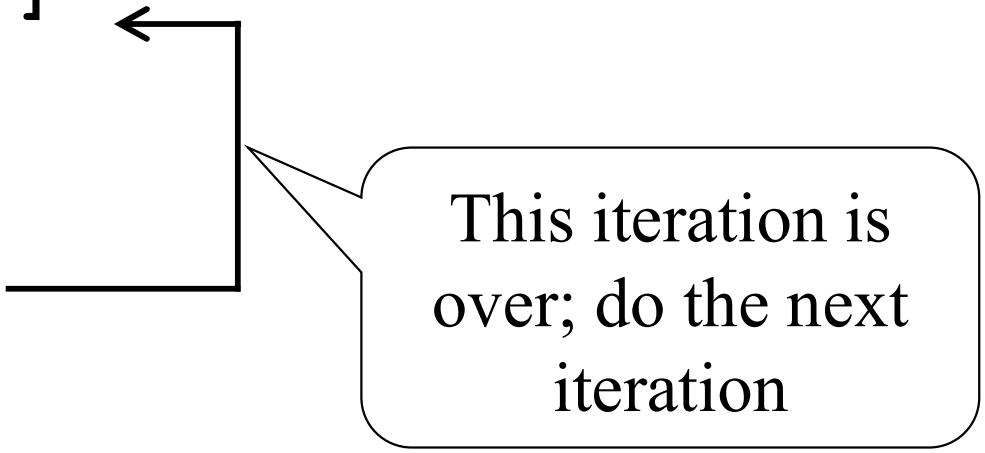
THE BREAK COMMAND

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



THE CONTINUE COMMAND

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



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
```

Shell Programming

- Conditional Statement - case
- Syntax:

```
case  $value  in
        val1)  command1
               command2;;
        val2)  command3;;
        ....
        *)   command4;;
esac
```

- The statement matches an expression for more than one alternative, and permits multi-way branching.

Case example

```
echo -e  "1.List of files\n
2.No. of processes\n
3.Today's date\n
4.Logged users\n
5.exit\n"
echo "Enter your choice"
read ch
case $ch in
1)ls ;;
2)ps ;;
3)date ;;
4)who ;;
5)exit ;;
*) echo "Wrong choice, enter again“
esac
```

2. The case...esac (Cont.)

- **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."
   ;;
esac
```

- **This will produce following result:**

  New Zealand is famous for kiwi.

SHELL PROGRAMMING

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

DONE !