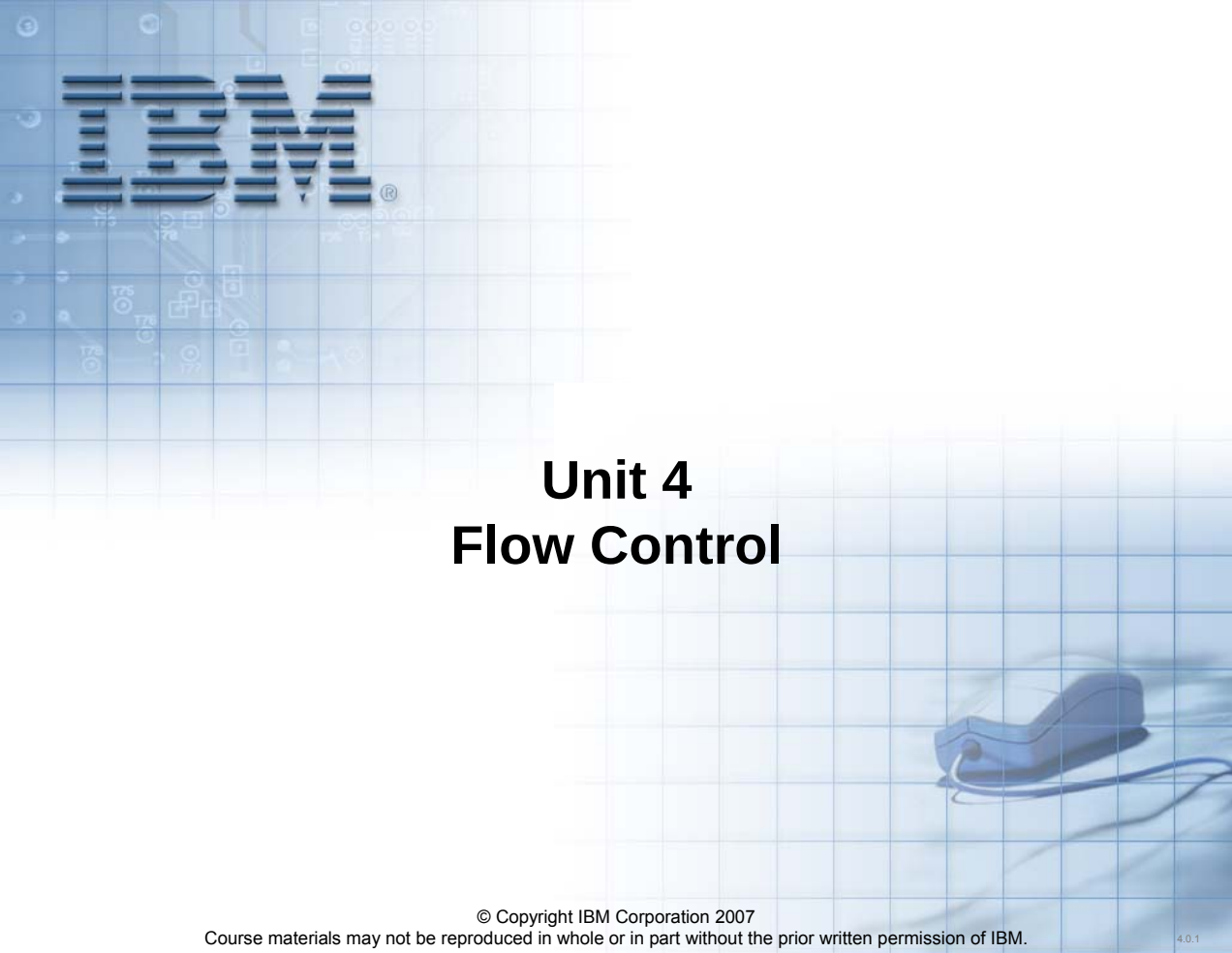


Unit 4
Flow Control

Unit Objectives

After completing this unit, you should be able to:

- Generate the **`if - then - else`** construct
- Generate conditional loops with **`until`** and **`while`**
- Understand specific value iteration with **`for`**
- Use multiple choice pattern matching with **`case`**
- Use the **`select`** command for menus
- Use **`break`** and **`continue`** in Loops
- Identify Doing Nothing – the **`null`** Command

The Simple *if - then - else* Construct

```
if expression1
then
   commands to be executed if expression1 is
      true

else
   commands to be executed if all expressions
      are false
fi
```

```
if   [[ "$UID" -lt "100" ]]
then
   export PS1="# "
else
   export PS1="$ "
fi
```

The Full *if - then - else* Construct

```
if [[ "$1" = "-a" ]]
then
  author=y ; control=n
  version=n
elif "$1" = "-c"
then
  author=n ; control=y
  version=n
elif "$1" = "-v"
then
  author=n ; control=n
  version=y
else
  author=n ; control=n
  version=n
fi      < /usr/local/defaults
```

if Example

Here is a simple if construct:

```
#!/usr/bin/ksh
# Usage: goodbye username
#
if [[ $# -ne 1 ]]
then
      print  "Usage is:  goodbye username"
      print  "Please try again."
      exit 1
fi
rmuser $1
print "O.K., $1 is removed."
```

When we run "goodbye", this is what we get ...

```
$ goodbye
Usage is: goodbye username
Please try again.
$ goodbye pete
O.K., pete is removed.
$ _
```

Conditional Loop Syntax

```
until cc prog.c
do
        vi prog.c
done
```

```
while [ "$x" -lt 3 ]
do
  lsps -a >> ./statfile
  df >> ./statfile
  let x=x+1
done    #  should be "< ./statfile" here
```

while true Example

The Script "forever" is a tough cookie!

```
#!/usr/bin/ksh
# An endless loop with a trap for INT QUIT TSTP
trap 'print "hasta la vista - baby!"' 2 3 18
while true
do
        print  "I'll be back."
        sleep 10
done
```

```
$  forever
I'll be back.
I'll be back.
I'll be back.
```

every ten seconds
the script speaks!

```
<Ctrl-c>
```

an attempt to stop it...

```
hasta la vista - baby!
I'll be back.
I'll be back.
```

invokes the trap, and
it carries on.

for Loop Syntax

```
for identifier in word1 word2 ...
do
  commands using $identifier
  more commands
done
```

```
for identifier    # equivalent to: for identifier in "$@"
do
  commands using $identifier which takes values from the
    positional parameters
done    #  optional < filename
```

for - in Loop Example

Here we have a quick tidy-up to delete files:

```
$  for  varfile  in  *.tmp
>  do
>   rm -f $varfile
>  done
$  _
```

Why use the option **`-f`** ?
What else could be **`test`**ed?

for Loop Example

The sample Script "getprice.ksh" will look up the price list:

```
#!/usr/bin/ksh
# getprice.ksh - select price from "pricelist" file
# for each item entered on the command line
# Usage: getprice item1 item2 ...
#
for item
do
        grep -i "$item" /home/cashier/pricelist
done


$ getprice.ksh  "Shock Absorbers"  "Air Filter"
Front Shock Absorbers      49.99
Rear Shock Absorbers       59.99
Air Filter                 10.99
$ _
```

Arithmetic for Loop

The arithmetic for loop is available in ksh93 and bash.

```
for (( initialize; test; increment ))
do
  commands
done
```

Example:

```
for (( num=0; num <5; num++ ))
do
  mv file${num} file${num}.bkup
done
```

The case Statement

```
case word in
( pattern1 | pattern2 | ... )
        action      ;;
(*)     default     ;;
esac
```

```
case $identifier in
(pattern1)                  command1
                            more_commands ;;
(pattern2 | pattern3)       commands    ;;
(*)                         commands    ;;
esac
```

case Code Example

A guessing game of sorts:

```
#!/usr/bin/ksh
# Usage:  match string
# To see how lucky you are feeling today

case "$1" in
   Ace )        print "You are really close."    ;;
   King )       print "Missed it by that much."  ;;
   Queen )      print "Finally!"                 ;;
   Jack )       print "Maybe next time."         ;;
   Ten|10 )     print "Getting closer."          ;;
   * )          print "Guess again."             ;;

esac
```

case Code Output

A casino dealer in the making?

```
$ match Three
Guess again.

$ match Jack
Maybe next time.

$ match Ace
You are really close.

$ match King
Missed it by that much.

$ match Queen
Finally!
```

Mini Quiz

1. True or False. There can be any number of **`elif`** statements in an **`if – then – else`** construct.

2. How does one redirect for the whole of an **`until`** or **`while`** loop?

3. True or False. The statement: "**`for identifier`**" takes its input from positional parameters.

The Shell select Syntax

```
select identifier in word1  word2  ...
do
  commands using $identifier usually containing a case
    statement
done
```

```
select identifier   # equivalent to: select identifier in "$@"
do
  commands using $identifier from positional parameters
    usually containing a case statement
done
```

select Code Example

To help identify animals we have a "barn.ksh" Shell Script:

```
#!/usr/bin/ksh
# usage: barn.ksh
PS3="Pick an animal: "
select  animal  in  cow  pig  dog  quit
do
        case $animal in
        (cow)   print "Moo"
                ;;
        (pig)   print "Oink"
                ;;
        (dog)   print "Woof"
                ;;
        (quit)  exit
                ;;
        ('')     print "Not in the barn"
                ;;
        esac
done
```

select Output Example

Running "barn.ksh" we can choose an animal to examine ...

```
$ barn.ksh
1) cow
2) pig
3) dog
4) quit
Pick an animal: 1
Moo
Pick an animal: 2
Oink
Pick an animal: 3
Woof
Pick an animal: 8
Not in the barn
Pick an animal: 4
$
```

Do you think setting PS3 to "Pick an animal" was a good choice?

More on Select

- In the previous example, the selected choice (for example **`cow`**) was stored in **`$animal`**, however, the input from the user was stored in **`$REPLY`**

- Using the **`$REPLY`** variable makes the select syntax a bit more flexible as seen on the next page

- In ksh93, the **`TMOUT`** variable can be set to a number of seconds. The select loop will timeout if no input is received within the **`TMOUT`** seconds set.

Select Example Using $REPLY

```
#!/usr/bin/ksh
# usage: barn.ksh
PS3 = "Pick an animal:"
Select animal in cow pig dog quit
do
  case $REPLY in
  cow|COW)          print "Moo"          ;;
  pig|PIG)          print "Oink"         ;;
  dog|DOG)          print "Woof"         ;;
  quit|QUIT)        exit                 ;;
  *)                print "Not in the barn"       ;;
  esac
done
```

exit the Loop

In the Korn shell script /usr/sbin/snap

```
...
if [ "$badargs" = n ]
then
  if [ "$found" = y ]
  then
    if [ -r "$destdir/$component/$component.snap" ]
    then
    more $destdir/$component/$component.snap
    else
    echo "^Gsnap:  $destdir/$component/$component.snap not found"
    exit 25
    fi
  fi
else
    usage
    exit 26
fi  ...
```

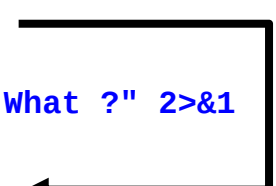
break the Loop

The **break** command jumps out of **do . . . done** loops:

- Exits from the smallest enclosing loop
- Jumps out a specified **number** of layers/loops

```
  break  number
```

```
select  choice  in  Backup  Restore  Quit
do
  case $choice in
  (Backup)  find . -print|backup -iqf /dev/rfd0
  ;;
  (Restore) restore -xqf /dev/rfd0
  ;;
  (Quit)    break
  ;;
  ('')      print "What ?" 2>&1
  ;;
  esac
done
```



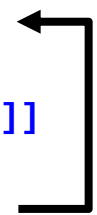
continue the Loop

The **continue** command begins the next iteration of a **do . . . done** loop:

- Starts at the top of the smallest enclosing loop
- Begins again a specified *number* of layers/loops out

```
continue  number
```

```
$ for File in *
> do
> if [[ -d $File ]]
> then
>       continue
> fi
> file $File
> done
$ _
```



null Logic

Sometimes you require a command, but you don't actually want to do anything – a **NULL** command

```
:          # a COLON character
```

For example:

```
sys_call parameter1 parameter2
if  [[  $?  -eq  0  ]]
then
     # Debug slot
     :
else
    print $0: Error: command failed
    exit $ERRNO
fi
```

without the null command ":" this would be illegal syntax

Program Logic Constructs Example

Here's a Script to delete empty files:

```
#!/usr/bin/ksh
# Usage: delfile file1 file2 ...
while [[ $# -gt 0 ]]
do
        if [[ -f "$1" ]]
        then
                if [[ ! -s "$1" ]]
                then
                        rm "$1" && print "$1" deleted
                else
                        print "$1" not deleted 2>&1
                fi
        elif [[ -d "$1" ]]
        then
                print "$1" is a directory
        else
                print "$1" is a special file
        fi
        shift
done
```

Checkpoint (1 of 2)

1. What is wrong with this fragment of shell script?

```
if [ "$x" -eq 5 ]
then
      echo $x
elif [ "$x" -eq 3 ]
else
      echo "x is only 3"
      exit
fi
```

2. What is the fundamental difference between a **`while`** and an **`until`** construct?
3. How could we write an endless loop?
4. What syntax would we use to perform a loop a finite number of times, resetting an identifier (variable) each time the loop goes through?

Checkpoint (2 of 2)

5. Which construct is best suited to allow conditional processing, based on pattern matching?
6. What would the following lines produce?

```
select word in To be or not to be
do
        :
done
```

7. Which construct is best used within the previous **do-done**? block?
8. How can we terminate one iteration of a loop and commence the next?
9. How can we abruptly terminate all iterations of a loop but continue further processing in a shell script?

Unit Summary

- The **`if – then – else`** construct
- Conditional loops with **`until`** and **`while`**
- Specific value iteration with **`for`**
- Multiple choice pattern matching with **`case`**
- The **`select`** command for menus
- Leaving loops – **`exit`** and **`break`**
- Beginning again – **`continue`**
- Doing nothing — the **`null`** command – **`:`**