Chapter One

- echo
- echo "\c"
- read
- if-then-fi
- [=, !=]
- nested if statements
- else
- elif
- shell variables
- case-in-esac
- ;;
- case *) default option
- double-quotes

Menus are one of the simplest shell scripts. All they have to do is display some options, ask the user to select one and run a command based on the user input. On the facing page are three simple menus. Whilst they will all appear to work the same, they have been written in different ways to illustrate various *if-then-else* constructs supported by shell script. The fourth position shows how the menus look when executed.

echo

1 *Menu1* starts with several *echo* commands. *Echo* is the shell's primary printing mechanism and as it's name suggests, *echo* simply prints whatever is passed to it within the *double-quotes*. *Echo* automatically adds a newline to the printed text.

Thus, the first five lines of *menu1* display the three menu options with a blank line above and below to make the menu look better when executed:

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
```

will produce the following output

```
1 - Who is logged on
2 - Disk space
3 - Date and time
```

2 Note that the *echo* command with no arguments only prints an empty line; *echo ""* is unnecessary and generally not used.

echo "\c"

3 The sixth *echo* line displays a prompt, so that the user can enter a selection. Here, *echo* is not required to add a newline to the end of the text, which will mean that the cursor remains next to the colon. Again this is aesthetic.

To prevent *echo* from adding a newline, the text to be printed should end with "\c". This is classed as an *escape sequence*, as it has a specific meaning to *echo.*

For example, the diagram below shows the effect of the *echo "Select"* line with and without the "\c":

```
echo "Select: \c"
Select: _
```

cursor after text

```
echo "Select: "
Select:
_
```

cursor on next line

Some shells do not support *echo "\c"*. If the shell you are using does not behave correctly, see Appendix B for shell compatibilities and remedial actions.

read

4 The *read* command, on the next line, reads a single line of input from the user and assigns it to the variable specified; in this instance INPUT. The variable INPUT will be created if it does not already exist and should it exist, the contents will be overwritten.

menu1

```
1  echo
   echo "1 - Who is logged on"
   echo "2 - Disk space"
   echo "3 - Date and time"
2  echo
3  echo "Select: \c"
4  read INPUT

   if [ "$INPUT" = "1" ]
   then
       who -q
   fi

   if [ "$INPUT" = "2" ]
   then
       df -k
   fi

   if [ "$INPUT" = "3" ]
   then
       date
   fi
```

menu2

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
else
    if [ "$INPUT" = "2" ]
    then
        df -k
    else
        if [ "$INPUT" = "3" ]
        then
            date
        fi
    fi
fi
```

menu3

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
elif [ "$INPUT" = "2" ]
then
    df -k
elif [ "$INPUT" = "3" ]
then
    date
fi
```

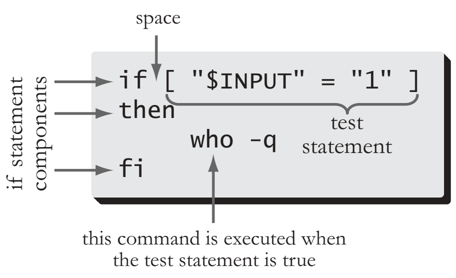
```
1 - Who is logged on
2 - Disk space
3 - Date and time

Select: 3
Mon  8 Oct 2001 GMT 19:06
```

1 Having listed the available options and obtained a response from the user, *menu1* now processes the user input.

test statement if, then and fi

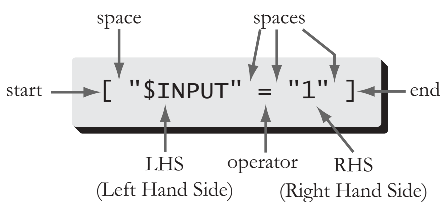
Menu1 uses three independent *if* statements; one for each option. The code after the *if* is called the *test* statement. If the *test* statement is true, the code between the *then* and the *fi* is executed. The *test* statement is examined separately below.

The *then* and *fi* are part of the *if* statement. *Then* is effectively a noise word (no purpose other than to make the code read better), though it is part of the syntax and therefore obligatory. *Fi* terminates the *if* statement. For every *if* there must be a *then* and a *fi*.



```
if [ "$INPUT" = "1" ]
then
     who -q
fi
```

The *test* statement is bounded by square brackets ([ ]). Each component of the test statement must be separated from it's neighbour by a space. Also, note that in order to access the contents of the INPUT variable, it must be preceded by a "$". Variable assignment and usage is covered later in this chapter.



```
[ "$INPUT" = "1" ]
```

equals

The *test* statement illustrated here is divided into three main components; the Left Hand Side (LHS), the operator and the Right Hand Side (RHS). The operator here is the equals sign and this instructs the *test* statement to check whether the LHS is the same as the RHS.

menu1

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
fi

if [ "$INPUT" = "2" ]
then
    df -k
fi

if [ "$INPUT" = "3" ]
then
    date
fi
```

menu2

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
else
    if [ "$INPUT" = "2" ]
    then
        df -k
    else
        if [ "$INPUT" = "3" ]
        then
            date
        fi
    fi
fi
```

menu3

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
elif [ "$INPUT" = "2" ]
then
    df -k
elif [ "$INPUT" = "3" ]
then
    date
fi
```

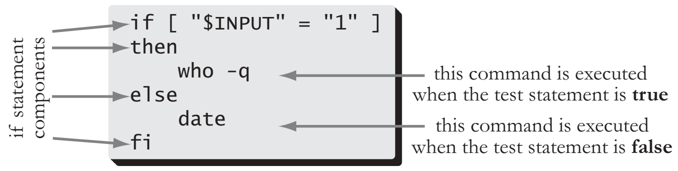
```
1 - Who is logged on
2 - Disk space
3 - Date and time

Select: 3
Mon  8 Oct 2001 GMT 19:06
```

Menu1 is inefficient because each of the three *if* statements are processed every time *menu1* is executed. The flow diagram next to *menu1*, opposite, shows that even after option 1 has been matched and the *who -q* command executed, the script still checks for options 2 and then for 3.

To make *menu1* more efficient, it has been rewritten as *menu2*, so that once the correct option has been found, no more checks are made. The flow diagram next to *menu2* demonstrates how after option 1 has been selected and *who -q* is executed, the script skips the rest of the checks and terminates.

else

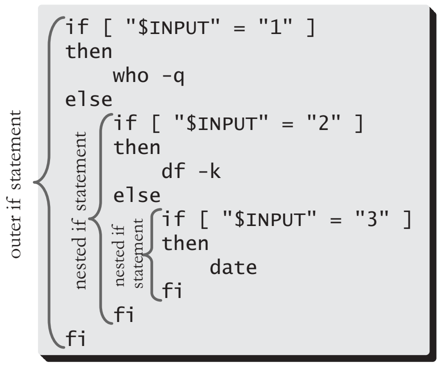
Menu2 achieves this greater efficiency by using the *else* component of the *if* statement. An *else* component means that there are two separate sections of code, only one of which will be executed. Any code between the *then* and the *else* is executed if the *test* statement is *true* and any code between the *else* and the *fi* is executed if the *test* statement is false.


if statement components
this command is executed when the test statement is **true**
this command is executed when the test statement is **false**

```
if [ "$INPUT" = "1" ]
then
    who -q
else
    date
fi
```

nested if statements

Menu2 uses three *nested if* statements. A *nested if* statement is an *if* statement that resides inside another *if* statement. The layout of *menu2* also shows how indentation can be used to make shell scripts more readable. For example, at the bottom of *menu2* are three *fi*'s. The relative levels of indentation should indicate which *fi* belongs to which *if*.

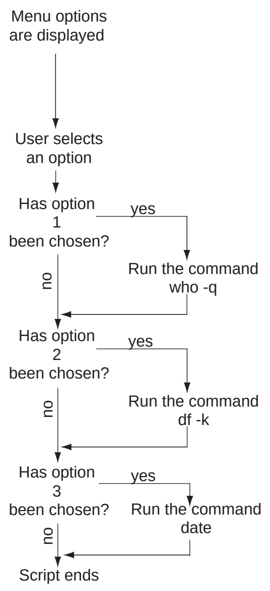


```
if [ "$INPUT" = "1" ]
then
    who -q
else
    if [ "$INPUT" = "2" ]
    then
        df -k
    else
        if [ "$INPUT" = "3" ]
        then
            date
        fi
    fi
fi
```

Menu2 is rather messy, though. If a further six options were added to menu, the indentation would become somewhat excessive. It is also hard to pick out the menu commands from the rest of the programming text.

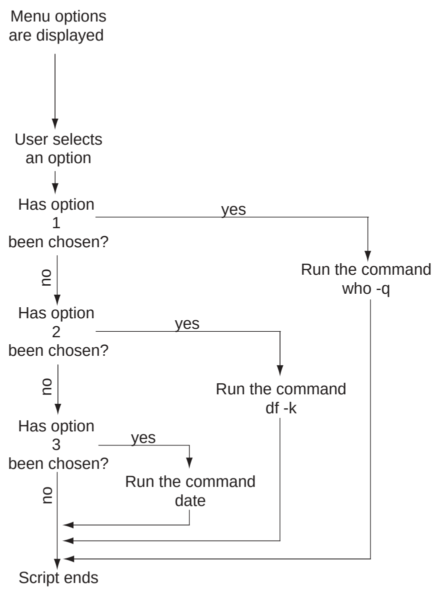
menu1

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
fi

if [ "$INPUT" = "2" ]
then
    df -k
fi

if [ "$INPUT" = "3" ]
then
    date
fi
```



menu2

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
else
    if [ "$INPUT" = "2" ]
    then
        df -k
    else
        if [ "$INPUT" = "3" ]
        then
            date
        fi
    fi
fi
```



1 elif *Menu3* uses *elifs* to overcome the layout problems encountered in *menu2*. *Menu3* works identically to *menu2* but the *else-if* sections have been combined into *elif*'s. This makes *menu3* a much shorter (and narrower) script than *menu2*. *Menu3* is also slightly more readable than *menu2*. Note, that the use of *elif*s means that *menu3* contains only a single *fi*, thus converting the three nested *if* statements in *menu2* into a single, multi-stage *if* statement.

```
if [ "$INPUT" = "1" ]
then
    who -q
else
    if [ "$INPUT" = "2" ]
    then
        df -k
    fi
fi
```

can be condensed using elif's to

```
if [ "$INPUT" = "1" ]
then
    who -q
elif [ "$INPUT" = "2" ]
then
    df -k
fi
```

2 double-quotes In all the shell code examined to date, the role of the *double-quotes* has been largely ignored. Understanding the use and effect of *double-quotes* within shell scripting is a significant step in writing trouble-free shell scripts.

In pure programming terms, nearly all of the *double-quotes* in the menus 1-3 are unnecessary. On the lower half of the opposite page is *menu2q*; *menu2* with all the *double-quotes* removed. *Menu2q* works identically to it's quoted equivalent, *menu2*, except for two main errors.

3 echo \c The first error is that the *echo Select \c* line no longer works as expected. This is because the \c needs to be enclosed in *double-quotes* in order to be passed to *echo*, as an unquoted *backslash* has special significance to the shell. Here, the unquoted *backslash* is stripped out by the shell, *echo* prints an extra “c+newline” and the cursor is displayed on the wrong line (see bottom right, opposite).

4 For this reason, *echo* statements will generally use *double-quotes* to ensure the output is not corrupted by the unwanted attentions of the shell. Besides, *echo* statements simply look more natural with *double-quotes*.

5 Next, to see the second error, run *menu2q* and press <Return> when prompted. *Menu2q* will generate an error where *menu2* would not have (bottom right example, opposite). This is due to that way that the shell code is parsed. In shell script, variables are substituted before the code is interpreted, rather than as an integral part of the interpretation. So when the *if* statement for option 1 is parsed with an empty $INPUT, the shell will try to execute the following, which is erroneous and will result in an error message.

```
if [ $INPUT = 1 ]
```

if $INPUT is empty will evaluate to

```
if [ = 1 ]
```

As a result, it is advisable that any variables used in *test* statements should be enclosed in *double-quotes*.

menu3

```
    echo
    echo "1 - Who is logged on"
    echo "2 - Disk space"
    echo "3 - Date and time"
    echo
    echo "Select: \c"
    read INPUT

    if [ "$INPUT" = "1" ]
    then
        who -q
[1] elif [ "$INPUT" = "2" ]
    then
        df -k
[1] elif [ "$INPUT" = "3" ]
    then
        date
    fi
```

menu2

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
else
    if [ "$INPUT" = "2" ]
    then
        df -k
    else
        if [ "$INPUT" = "3" ]
        then
            date
        fi
    fi
fi
```

menu2q

```
[2] echo
    echo 1 - Who is logged on
    echo 2 - Disk space
    echo 3 - Date and time
    echo
[3] echo Select: \c
    read INPUT

[5] if [ $INPUT = 1 ]
    then
        who -q
    else
        if [ $INPUT = 2 ]
        then
            df -k
        else
            if [ $INPUT = 3 ]
            then
                date
            fi
        fi
    fi
```

```
1 - Who is logged on
2 - Disk space
3 - Date and time

Select: c  <- unwanted character [3]
<Return> [4]
test: argument expected [5]
```

The next stage of development for these menus is to enhance them to include exception handling; that is, for when the user selects an invalid option.

Menu1d is a new version of *menu1*. Code new for *menu1d* is marked by the circles.

The script works by recording whether a user selected a valid option by setting a variable $FOUND. This is performed within the *if* statements.

1 != not equals

At the end of the script a new *if* statement checks $FOUND and if it is not equal (!=) to YES an error message is displayed. $FOUND will only be set to "YES" if a valid option was selected.

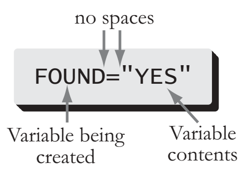
menu1d

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
    FOUND="YES"
fi

if [ "$INPUT" = "2" ]
then
    df -k
    FOUND="YES"
fi

if [ "$INPUT" = "3" ]
then
    date
    FOUND="YES"
fi

if [ "$FOUND" != "YES" ]
then
    echo "Invalid selection"
fi
```

2 creating shell variables

Note that there is no specific command required to create shell variables. Any line that starts with a word, followed immediately by an equals sign will cause a variable to be created. The variable will be assigned the value after the equals sign.

Shell variable names have to start with a letter and can only contain letters, numbers and underscores; no spaces. Whilst shell variables can use upper or lowercase letters, uppercase variable names are normally used to help distinguish them from the rest of the shell code.



Menu2 & *menu3* are easy to convert to include default handling. Both scripts require an *else* statement with a suitable message after the *date* command.

menu2d

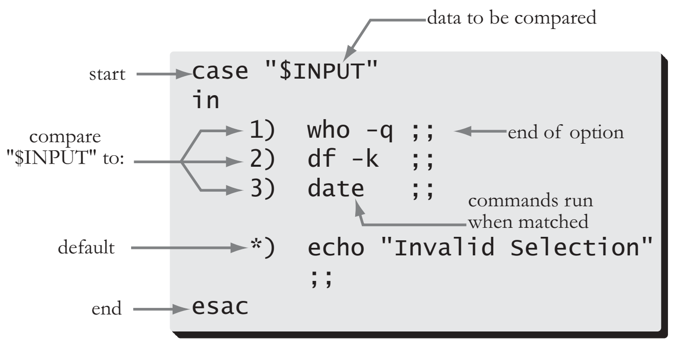
```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
else
    if [ "$INPUT" = "2" ]
    then
        df -k
    else
        if [ "$INPUT" = "3" ]
        then
            date
        else
            echo "Invalid selection"
        fi
    fi
fi
```

menu3d

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
elif [ "$INPUT" = "2" ]
then
    df -k
elif [ "$INPUT" = "3" ]
then
    date
else
    echo "Invalid selection"
fi
```

1 case On the opposite page is *menu4*, which introduces a new command: *case*. A *case* statement works somewhat like the *if-elif* construct in *menu3d* in that a single value is compared against a list and some code executed as a result; they are just laid out a little different.

By comparing the location of the various components of the *case* statement with *menu3d*, opposite, it should be possible to identify the key areas. *Menu3d* and *menu4* work identically.



```
case "$INPUT"
in
     1)  who -q ;;
     2)  df -k  ;;
     3)  date   ;;

     *)  echo "Invalid Selection"
         ;;
esac
```

The following points should be noted regarding *case* statements:

2 • The data to be compared ($INPUT as usual) is located after the keyword *case* and before the keyword *in*. It should be enclosed in *double-quotes*.

3 • The *in-esac* pair delimits the body of the *case* statement.

4 • Options 1, 2 & 3 are listed opposite the commands to be executed when matched.

• There should not be a space between the option and the close bracket or the *case* statement will generate a “syntax error”.

5 • The ;; separates the individual *case* options.

• After a match, the code between the “)” and the “;;” is executed.

6 • The final asterisk (*) is used for default handling. The asterisk means literally "match anything" and as such should always be the last option in a *case* statement.

• *Case* statements do not have to have a default handler.

• The *case* statement will process only the first option that matches
7 $INPUT.

• The ;; can be on a different line to the option.

menu4

```
  echo
  echo "1 - Who is logged on"
  echo "2 - Disk space"
  echo "3 - Date and time"
  echo
  echo "Select: \c"
  read INPUT

1 case "$INPUT"2
3 in
4     1)  who -q ;;5
      2)  df -k  ;;
      3)  date   ;;

6     *)  echo "Invalid selection"
7         ;;
3 esac
```

menu3d

```
echo
echo "1 - Who is logged on"
echo "2 - Disk space"
echo "3 - Date and time"
echo
echo "Select: \c"
read INPUT

if [ "$INPUT" = "1" ]
then
    who -q
elif [ "$INPUT" = "2" ]
then
    df -k
elif [ "$INPUT" = "3" ]
then
    date
else
    echo "Invalid selection"
fi
```

Case statements are very compact, but are remarkably flexible. *Menu4* is much easier to read than *menu3d* owing to the lack of repeated code.

Case statements are used extensively throughout this book. They should be studied carefully as they provide a significant level of functionality not found in any other shell command.