7.3. Other Comparison Operators

A *binary* comparison operator compares two variables or quantities. *Note that integer and string comparison use a different set of operators.*

integer comparison

-eq

is equal to

```
if [ "$a" -eq "$b" ]
```

-ne

is not equal to

```
if [ "$a" -ne "$b" ]
```

-gt

is greater than

```
if [ "$a" -gt "$b" ]
```

-ge

is greater than or equal to

```
if [ "$a" -ge "$b" ]
```

-lt

is less than

```
if [ "$a" -lt "$b" ]
```

-le

is less than or equal to

```
if [ "$a" -le "$b" ]
```

<

is less than (within double parentheses)

```
(("$a" < "$b"))
```

<=

is less than or equal to (within double parentheses)

(("$a" <= "$b"))

>

is greater than (within double parentheses)

(("$a" > "$b"))

>=

is greater than or equal to (within double parentheses)

(("$a" >= "$b"))

string comparison

=

is equal to

if ["$a" = "$b"]

Note the whitespace framing the =.

if ["$a"="$b"] is *not* equivalent to the above.

==

is equal to

if ["$a" == "$b"]

This is a synonym for =.

The == comparison operator behaves differently within a double-brackets test than within single brackets.

```
[[ $a == z* ]]   # True if $a starts with an "z" (pattern matching).
[[ $a == "z*" ]] # True if $a is equal to z* (literal matching).

[ $a == z* ]     # File globbing and word splitting take place.
[ "$a" == "z*" ] # True if $a is equal to z* (literal matching).

# Thanks, Stéphane Chazelas
```

!=

is not equal to

if ["$a" != "$b"]

This operator uses pattern matching within a [[...]] construct.

<

is less than, in ASCII alphabetical order

if [["$a" < "$b"]]

if ["$a" \< "$b"]

Note that the "<" needs to be escaped within a `[ ]` construct.

\>

is greater than, in ASCII alphabetical order

`if [[ "$a" > "$b" ]]`

`if [ "$a" \> "$b" ]`

Note that the ">" needs to be escaped within a `[ ]` construct.

See Example 27-11 for an application of this comparison operator.

-z

string is *null*, that is, has zero length

```
 String=''   # Zero-length ("null") string variable.

if [ -z "$String" ]
then
  echo "\$String is null."
else
  echo "\$String is NOT null."
fi     # $String is null.
```

-n

string is not *null.*

The **`-n`** test requires that the string be quoted within the test brackets. Using an unquoted string with *! -z*, or even just the unquoted string alone within test brackets (see Example 7-6) normally works, however, this is an unsafe practice. *Always* quote a tested string. [1]

Example 7-5. Arithmetic and string comparisons

```
#!/bin/bash

a=4
b=5

#  Here "a" and "b" can be treated either as integers or strings.
#  There is some blurring between the arithmetic and string comparisons,
#+ since Bash variables are not strongly typed.

#  Bash permits integer operations and comparisons on variables
#+ whose value consists of all-integer characters.
#  Caution advised, however.

echo

if [ "$a" -ne "$b" ]
then
  echo "$a is not equal to $b"
  echo "(arithmetic comparison)"
fi

echo
```

```
if [ "$a" != "$b" ]
then
  echo "$a is not equal to $b."
  echo "(string comparison)"
  #     "4"  != "5"
  # ASCII 52 != ASCII 53
fi

# In this particular instance, both "-ne" and "!=" work.

echo

exit 0
```

Example 7-6. Testing whether a string is *null*

```
#!/bin/bash
#  str-test.sh: Testing null strings and unquoted strings,
#+ but not strings and sealing wax, not to mention cabbages and kings . . .

# Using   if [ ... ]

# If a string has not been initialized, it has no defined value.
# This state is called "null" (not the same as zero!).

if [ -n $string1 ]    # string1 has not been declared or initialized.
then
  echo "String \"string1\" is not null."
else  
  echo "String \"string1\" is null."
fi                    # Wrong result.
# Shows $string1 as not null, although it was not initialized.

echo

# Let's try it again.

if [ -n "$string1" ]  # This time, $string1 is quoted.
then
  echo "String \"string1\" is not null."
else  
  echo "String \"string1\" is null."
fi                    # Quote strings within test brackets!

echo

if [ $string1 ]       # This time, $string1 stands naked.
then
  echo "String \"string1\" is not null."
else  
  echo "String \"string1\" is null."
fi                    # This works fine.
# The [ ... ] test operator alone detects whether the string is null.
# However it is good practice to quote it (if [ "$string1" ]).
#
# As Stephane Chazelas points out,
#    if [ $string1 ]    has one argument, "]"
#    if [ "$string1" ]  has two arguments, the empty "$string1" and "]"


echo
```

```
string1=initialized

if [ $string1 ]       # Again, $string1 stands unquoted.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                    # Again, gives correct result.
# Still, it is better to quote it ("$string1"), because . . .


string1="a = b"

if [ $string1 ]       # Again, $string1 stands unquoted.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                    # Not quoting "$string1" now gives wrong result!

exit 0   # Thank you, also, Florian Wisser, for the "heads-up".
```

Example 7-7. ***zmore***

```
#!/bin/bash
# zmore

# View gzipped files with 'more' filter.

E_NOARGS=85
E_NOTFOUND=86
E_NOTGZIP=87

if [ $# -eq 0 ] # same effect as:  if [ -z "$1" ]
# $1 can exist, but be empty:  zmore "" arg2 arg3
then
  echo "Usage: `basename $0` filename" >&2
  # Error message to stderr.
  exit $E_NOARGS
  # Returns 85 as exit status of script (error code).
fi

filename=$1

if [ ! -f "$filename" ]   # Quoting $filename allows for possible spaces.
then
  echo "File $filename not found!" >&2   # Error message to stderr.
  exit $E_NOTFOUND
fi

if [ ${filename##*.} != "gz" ]
# Using bracket in variable substitution.
then
  echo "File $1 is not a gzipped file!"
  exit $E_NOTGZIP
fi

zcat $1 | more

# Uses the 'more' filter.
# May substitute 'less' if desired.

exit $?   # Script returns exit status of pipe.
```

```
#  Actually "exit $?" is unnecessary, as the script will, in any case,
#+ return the exit status of the last command executed.
```

compound comparison

-a

logical and

exp1 -a exp2 returns true if *both* exp1 and exp2 are true.

-o

logical or

exp1 -o exp2 returns true if either exp1 *or* exp2 is true.

These are similar to the Bash comparison operators **&&** and ||, used within double brackets.

```
[[ condition1 && condition2 ]]
```

The **-o** and **-a** operators work with the test command or occur within single test brackets.

```
if [ "$expr1" -a "$expr2" ]
then
  echo "Both expr1 and expr2 are true."
else
  echo "Either expr1 or expr2 is false."
fi
```

But, as *rihad* points out:

```
[ 1 -eq 1 ] && [ -n "`echo true 1>&2`" ]   # true
[ 1 -eq 2 ] && [ -n "`echo true 1>&2`" ]   # (no output)
# ^^^^^^^ False condition. So far, everything as expected.

# However ...
[ 1 -eq 2 -a -n "`echo true 1>&2`" ]       # true
# ^^^^^^^ False condition. So, why "true" output?

# Is it because both condition clauses within brackets evaluate?
[[ 1 -eq 2 && -n "`echo true 1>&2`" ]]     # (no output)
# No, that's not it.

# Apparently && and || "short-circuit" while -a and -o do not.
```

Refer to Example 8-3, Example 27-17, and Example A-29 to see compound comparison operators in action.

Notes

[1] As S.C. points out, in a compound test, even quoting the string variable might not suffice. `[ -n "$string" -o "$a" = "$b" ]` may cause an error with some versions of Bash if `$string` is empty. The safe way is to append an extra character to possibly empty variables, `[ "x$string" != x -o "x$a" = "x$b" ]` (the "x's" cancel out).